IBM i

DB2 for i SQL Reference

7.1

IBM i

DB2 for i SQL Reference

7.1

Note

Before using this information and the product it supports, read the information in Appendix K, "Notices," on page 1823.

This edition applies to IBM i 7.1 (product number 5770–SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

Contents

About DB2 for i SQL Reference

About SQL reference

This book defines Structured Query Language (SQL) as supported by DB2® for IBM® i. It contains reference information for the tasks of system administration, database administration, application programming, and operation. This manual includes syntax, usage notes, keywords, and examples for each of the SQL statements used on the system.

Standards compliance

DB2 for i 7.1 conforms with IBM and Industry SQL Standards.

- *ISO/IEC FCD 9075-1:2011, Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework)*
- *ISO/IEC FCD 9075-2:2011, Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation)*
- *ISO/IEC FCD 9075-3:2008, Information technology - Database languages - SQL - Part 3: Call-Level Interface (SQL/CLI)*
- *ISO/IEC FCD 9075-4:2011, Information technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM)*
- *ISO/IEC FCD 9075-10:2008, Information technology - Database languages - SQL - Part 10: Object Language Bindings (SQL/OLB)*
- *ISO/IEC FCD 9075-11:2011, Information technology - Database languages - SQL - Part 11: Information and Definition Schemas (SQL/Schemata)*
- *ISO/IEC FCD 9075-14:2011, Information technology - Database languages - SQL - Part 14: XML-Related Specifications (SQL/XML)*
- *INCITS/ISO/IEC 9075-1:2011, Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework)*
- *INCITS/ISO/IEC 9075-2:2011, Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation)*
- *INCITS/ISO/IEC 9075-3:2008, Information technology - Database languages - SQL - Part 3: Call-Level Interface (SQL/SCLI)*
- *INCITS/ISO/IEC 9075-4:2011, Information technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM)*
- *INCITS/ISO/IEC 9075-10:2011, Information technology - Database languages - SQL - Part 10: Object Language Bindings (SQL/OLB)*
- *INCITS/ISO/IEC 9075-11:2011, Information technology - Database languages - SQL - Part 11: Information and Definition Schemas (SQL/Schemata)*
- *INCITS/ISO/IEC 9075-14:2011, Information technology - Database languages - SQL - Part 14: XML-Related Specifications (SQL/XML)*

For strict adherence to the standards, consider using the standards options. Standards options can be specified through the following interfaces:

Table 1. Standards Option Interfaces

SQL Interface	Specification
Embedded SQL	SQLCURRULE(*STD) parameter on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to set the SQLCURRULE values. (For more information about CRTSQLxxx commands, see Embedded SQL Programming.)
Run SQL Statements	SQLCURRULE(*STD) parameter on the Run SQL Statements (RUNSQLSTM) command. (For more information about the RUNSQLSTM command, see SQL Programming.)
Call Level Interface (CLI) on the server	SQL_ATTR_HEX_LITERALS connection attribute (For more information about CLI, see SQL Call Level Interfaces (ODBC).)
JDBC or SQLJ on the server using IBM Developer Kit for Java™	Translate Hex connection property object (For more information about JDBC and SQLJ, see IBM Developer Kit for Java.)
ODBC on a client using the IBM i Access Family ODBC Driver	Hex Parser Option in ODBC Setup (For more information about ODBC, see System i® Access.)
OLE DB on a client using the IBM i Access Family OLE DB Provider	Hex Parser Option Connection Object Properties (For more information about OLE DB, see System i Access.)
ADO .NET on a client using the IBM i Access Family ADO .NET Provider	HexParserOption in Connection Object Properties (For more information about ADO .NET, see System i Access.)
JDBC on a client using the IBM Toolbox for Java	Interpret SQL hexadecimal constants as binary data in JDBC Setup (For more information about JDBC, see System i Access.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java.)

Assumptions relating to examples of SQL statements

The examples of SQL statements shown in this guide assume the following.

- SQL keywords are highlighted.
- Table names used in the examples are the sample tables provided in the SQL Programming topic collection. Table names that are not provided in that appendix should use schemas that you create. You can create a set of sample tables in your own schema by issuing the following SQL statement:

  ```
  CALL QSYS.CREATE_SQL_SAMPLE ('your-schema-name')
  ```

- The SQL naming convention is used.
- For COBOL examples, the APOST and APOSTSQL precompiler options are assumed (although they are not the default in COBOL). Character-string constants within SQL and host language statements are delimited by apostrophes (').
- A collating sequence of *HEX is used.

Whenever the examples vary from these assumptions, it is stated.

How to read the syntax diagrams

The following rules apply to the syntax diagrams used in this book.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►— symbol indicates the beginning of the syntax diagram.

  The —► symbol indicates that the syntax is continued on the next line.

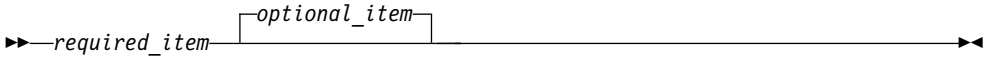
  The ►— symbol indicates that the syntax is continued from the previous line.

  The —►◄ symbol indicates the end of the syntax diagram.

  Diagrams of syntactical units start with the |— symbol and end with the —| symbol.
- Required items appear on the horizontal line (the main path).

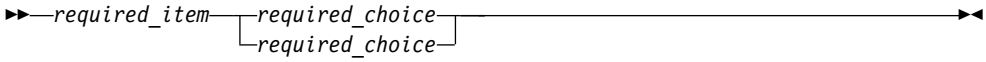
- Optional items appear below the main path.

  If an item appears above the main path, that item is optional, and has no effect on the execution of the statement and is used only for readability.

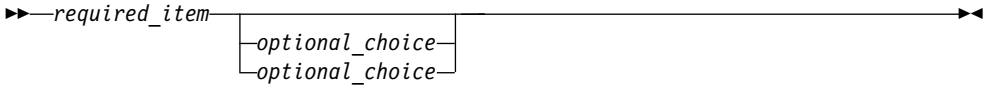


- If more than one item can be chosen, they appear vertically, in a stack.

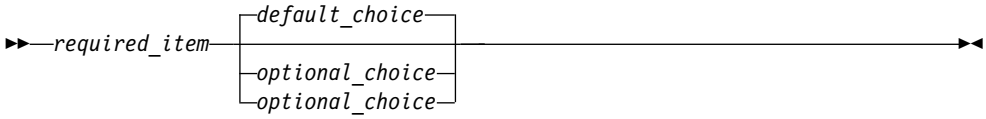
  If one of the items must be chosen, one item of the stack appears on the main path.



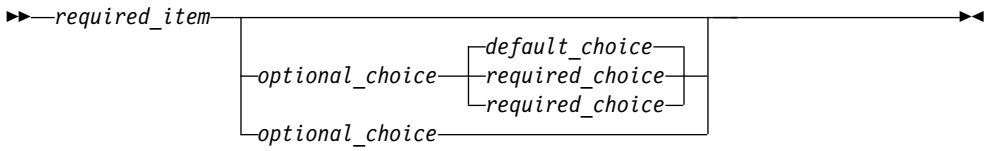
  If choosing one of the items is optional, the entire stack appears below the main path.



  If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.

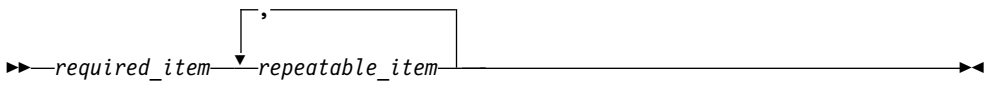


  If an optional item has a default when it is not specified, the default appears above the main path.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.

If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that the items in the stack can be repeated.

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.
- The syntax diagrams only contain the preferred or standard keywords. If nonstandard synonyms are supported in addition to the standard keywords, they are described in the **Notes** sections instead of the syntax diagrams. For maximum portability, use the preferred or standard keywords.
- Sometimes a single variable represents a larger fragment of the syntax. For example, in the following diagram, the variable parameter-block represents the whole syntax fragment that is labeled **parameter-block**:

parameter-block:

Conventions used in this book

This section specifies some conventions which are used throughout this manual.

Highlighting conventions

Bold	Indicates SQL keywords used in examples and when introducing descriptions involving the keyword.

Italics	Indicates one of the following: • Variables that represent items from a syntax diagram. • The introduction of a new term. • A reference to another source of information.

Conventions for describing mixed data values

When mixed data values are shown in the examples, the following conventions apply:

Convention	Meaning
$^{S}_{0}$	Represents the EBCDIC *shift-out* control character (X'0E')
$^{S}_{I}$	Represents the EBCDIC *shift-in* control character (X'0F')
sbcs-string	Represents a string of zero or more single-byte characters
dbcs-string	Represents a string of zero or more double-byte characters
'	Represents a DBCS apostrophe (EBCDIC X'427D')
G	Represents a DBCS G (EBCDIC X'42C7')

Conventions for describing mixed data values description

Conventions for describing mixed data values. Shift-out character represented by X' 0E', shift-in characters represented by X' 0F', single-byte characters represented by sbcs-string, double-byte characters represented by dbcs-string, DBCS apostrophe represented by EBCDIC X' 427D, and DBCS G represented by EBCDIC X' 42C7.

Conventions for describing Unicode data

When a specific Unicode UTF-16 code point is referenced, it can be expressed as U+*n*, where *n* is 4 to 6 hexadecimal digits. Leading zeros are omitted, unless the code point has fewer than 4 hexadecimal digits. For example, the following values are valid representations of a UTF-16 code point:

```
U+00001  U+0012  U+0123  U+1234  U+12345  U+123456
```

SQL accessibility

IBM is committed to providing interfaces and documentation that are easily accessible to the disabled community.

For general information about IBM's Accessibility support visit the Accessibility Center at http://www.ibm.com/able.

SQL accessibility support falls in two main categories.

- System i Navigator is graphical user interface to the IBM i operating system and DB2 for i. For information about the Accessibility features supported in Windows graphical user interfaces, see Accessibility in the Windows Help Index.
- Online documentation, online help, and prompted SQL interfaces can be accessed by a Windows Reader program such as the IBM Home Page Reader. For information about the IBM Home Page Reader and other tools, visit the Accessibility Center.

SQL accessibility

The IBM Home Page Reader can be used to access all descriptive text in this book, all articles in the SQL Information Center, and all SQL messages. Due to the complex nature of SQL syntax diagrams, however, the reader will skip syntax diagrams. Two alternatives are provided for better ease of use:

- Interactive SQL and Query Manager

  Interactive SQL and Query Manager are traditional file interfaces that provide prompting for SQL statements. These are part of the IBM DB2 Query Manager and SQL Development Kit for i. For more information about Interactive SQL and Query Manager, see the SQL Programming and Query Manager Use topics.
- SQL Assist

  SQL Assist is a graphical user interface that provides a prompted interface to SQL statements. This is part of System i Navigator. For more information, see the System i Navigator online help and the Information Center.

PDF file for SQL reference

Use this to view and print a PDF of this information.

To view or download the PDF version of this document, select SQL reference.

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF in your browser (right-click the link above).
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site

(http://get.adobe.com/reader/) .

What's new for IBM i 7.1

This topic highlights the changes made to this topic collection for IBM i 7.1.

The major new features covered in this book include:

- Three-part names in statements and aliases
- XML data type
- Global variables
- Array types in SQL procedures
- XML publishing functions
- MQ Series scalar functions
- BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT scalar functions
- Currently committed concurrent access resolution
- Consuming result sets in embedded SQL
- Expressions in the CALL statement
- REPLACE option for CREATE statements

- ALTER TABLE ADD COLUMN BEFORE
- Field procedures
- Removal of identity column and constraint restrictions on partition tables
- Encoded vector indexes INCLUDE of aggregate functions
- MERGE statement
- Inlining of some SQL scalar functions
- Parameter marker enhancements

What's new as of April 2016

- Additional formatting options for TIMESTAMP_FORMAT and VARCHAR_FORMAT scalar functions.
- Enhancements to ROUND and TRUNCATE scalar functions.
- Support for the EXTIND option in SET OPTION for SQL functions, procedures, and triggers.
- Required DB2 PTF Group level tracking added for static statements to SYSPROGRAMSTMTSTAT.

What's new as of October 2015

- LIMIT and OFFSET support for queries
- OVERLAY and LOCATE_IN_STRING scalar functions
- System name for CREATE VARIABLE

What's new as of April 2015

- REPLACE option for CREATE TABLE
- VARBINARY_FORMAT and VARCHAR_FORMAT_BINARY scalar functions
- Maximum parameters for SQL functions increased to 1024

What's new as of October 2014

- LPAD and RPAD scalar functions
- REGEXP_xxx scalar functions and REGEXP_LIKE predicate for regular expression handling
- Support for PIPE statement in SQL table functions

What's new as of April 2014

- A create table statement that has a single partitioning key column can have the range partitions defined in any order.
- SYSPACKAGESTMTSTAT and SYSPROGRAMSTMTSTAT catalog views

What's new as of October 2013

- Compound (dynamic) statement can group multiple statements into an executable routine
- CREATE TABLE AS and CREATE TABLE LIKE generate REFFLD information
- An RDB alias can be specified directly in an SQL statement using a three-part name
- Maximum index size increased to 1.7 terabytes

What's new as of February 2013

- Multiple event triggers including a new trigger event predicate

- Specify system name on CREATE TABLE, CREATE INDEX, CREATE VIEW, and DECLARE GLOBAL TEMPORARY TABLE
- BASE_TABLE table function

What's new as of October 2012

- Default values for procedure parameters
- Named parameters for procedure invocation
- Allow period separator for qualification of objects in system naming
- CREATE TABLE and DECLARE GLOBAL TEMPORARY TABLE where the new table and the select-statement reference different relational databases
- A new QAQQINI option, SQL_GVAR_BUILD_RULE, to influence receiving errors for variables not found at precompile or create routine time.

What's new as of April 2012

- Obfuscation for SQL procedures and SQL functions
- XMLTABLE table function
- INSERT where the target table and the select-statement reference different relational databases

What's new as of October 2011

- SYSPARTITIONINDEXDISK view

What's new as of April 2011

- CONNECT BY for hierarchical queries
- Program name for CREATE TRIGGER

How to see what's new or changed

To help you see where technical changes have been made, this information uses:

- The  image to mark where new or changed information begins.
- The  image to mark where new or changed information ends.

To find other information about what's new or changed this release, see the Memo to users.

Chapter 1. Concepts

This chapter provides a high-level view of concepts that are important to understand when using Structured Query Language (SQL). The reference material contained in the rest of this manual provides a more detailed view.

Relational database

A *relational database* is a database that can be perceived as a set of tables and can be manipulated in accordance with the relational model of data. The relational database contains a set of objects used to store, access, and manage data. The set of objects includes tables, views, indexes, aliases, user defined types, functions, procedures, sequences, variables, and packages.

There are three types of relational databases a user can access from a IBM i product.

system relational database
There is one default relational database on any IBM i product. The system relational database is always local to that IBM i product. It consists of all the database objects that exist on disk attached to the IBM i product that are not stored on independent auxiliary storage pools. For more information on independent auxiliary storage pools, see the System Management category of the IBM i Information Center.

The name of the system relational database is, by default, the same as the IBM i system name. However, a different name can be assigned through the use of the ADDRDBDIRE (Add RDB Directory Entry) command or IBM i Navigator.

user relational database
The user may create additional relational databases on a IBM i product by configuring independent auxiliary storage pools on the system. Each primary independent auxiliary storage pool is a relational database. It consists of all the database objects that exist on the independent auxiliary storage pool disks. Additionally, all database objects in the system relational database of the IBM i product to which the independent auxiliary storage pool is connected are logically included in a user relational database. Thus, the name of any schema created in a user relational database must not already exist in that user relational database or in the associated system relational database.

Although the objects in the system relational database are logically included in a user relational database, certain dependencies between the objects in the system relational database and the user relational database are not allowed:

- A view must be created into a schema that exists in the same relational database as its referenced tables and views, except that a view created into QTEMP can reference tables and views in the user relational database.
- An index must be created into a schema that exists in the same relational database as its referenced table.

- A trigger or constraint must be created into a schema that exists in the same relational database as its base table.
- The parent table and dependent table in a referential constraint must both exist in the same relational database.
- Any object in the system relational database can only reference functions, procedures, and types in the same system relational database. However, objects in the user relational database may reference functions, procedures, and types in the system relational database or the same user relational database. However, operations on such an object may fail if the other relational database is not available. For example, if a user relational database is varied off and then varied on to another system.

A user relational database is local to a IBM i product while the independent auxiliary storage pool is varied on. Independent auxiliary storage pools can be varied off on one IBM i product and then varied on to another IBM i product. Hence, a user relational databases may be local to a given IBM i product at one point in time and remote at a different point in time. For more information on independent auxiliary storage pools, see the System Management category of the IBM i Information Center.

The name of the user relational database is, by default, the same as the independent auxiliary storage pool name. However, a different name can be assigned through the use of the ADDRDBDIRE (Add RDB Directory Entry) command or System i Navigator.

remote relational database
Relational databases on other IBM i and non-IBM i products can be accessed remotely. These relational databases must be registered through the use of the ADDRDBDIRE (Add RDB Directory Entry) command or IBM i Navigator.

The database manager is the name used generically to identify the IBM i Licensed Internal Code and the DB2 for i portion of the code that manages the relational database.

Structured Query Language

Structured Query Language (SQL) is a standardized language for defining and manipulating data in a relational database. In accordance with the relational model of data, the database is perceived as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more base tables.

SQL statements are executed by a database manager. One of the functions of the database manager is to transform the specification of a result table into a sequence of internal operations that optimize data retrieval. This transformation occurs when the SQL statement is *prepared*. This transformation is also known as *binding*.

All executable SQL statements must be prepared before they can be executed. The result of preparation is the executable or *operational form* of the statement. The method of preparing an SQL statement and the persistence of its operational form distinguish *static* SQL from *dynamic* SQL.

Static SQL

The source form of a *static* SQL statement is embedded within an application program written in a host language such as COBOL, C, or Java. The statement is

prepared before the program is executed and the operational form of the statement persists beyond the execution of the program.

A source program containing static SQL statements must be processed by an SQL precompiler before it is compiled. The precompiler checks the syntax of the SQL statements, turns them into host language comments, and generates host language statements to call the database manager.

The preparation of an SQL application program includes precompilation, the preparation of its static SQL statements, and compilation of the modified source program.

Dynamic SQL

Programs containing embedded *dynamic* SQL statements must be precompiled like those containing static SQL, but unlike static SQL, the dynamic SQL statements are constructed and prepared at run time. The source form of the statement is a character or graphic string that is passed to the database manager by the program using the static SQL PREPARE or EXECUTE IMMEDIATE statement. A statement prepared using the PREPARE statement can be referenced in a DECLARE CURSOR, DESCRIBE, or EXECUTE statement. The operational form of the statement persists for the duration of the connection or until the last SQL program leaves the call stack.

SQL statements embedded in a REXX application are dynamic SQL statements. SQL statements submitted to the interactive SQL facility and to the Call Level Interface (CLI) are also dynamic SQL statements.

Extended Dynamic SQL

An extended dynamic SQL statement is neither fully static nor fully dynamic. The QSQPRCED API provides users with extended dynamic SQL capability. Like dynamic SQL, statements can be prepared, described, and executed using this API. Unlike dynamic SQL, SQL statements prepared into a package by this API persist until the package or statement is explicitly dropped. For more information, see the Database and File APIs information in the **Programming** category of the IBM i Information Center.

Interactive SQL

An interactive SQL facility is associated with every database manager. Essentially, every interactive SQL facility is an SQL application program that reads statements from a workstation, prepares and executes them dynamically, and displays the results to the user. Such SQL statements are said to be issued *interactively*.

The interactive facilities for DB2 for i are invoked by the STRSQL command, the STRQM command, or the Run SQL Script support of System i Navigator. For more information about the interactive facilities for SQL, see the SQL Programming and Query Manager Use  books.

SQL Call Level Interface and Open Database Connectivity

The DB2 Call Level Interface (CLI) is an application programming interface in which functions are provided to application programs to process dynamic SQL statements. DB2 CLI allows users of any of the ILE languages to access SQL

functions directly through procedure calls to a service program provided by DB2 for i. CLI programs can also be compiled using an Open Database Connectivity (ODBC) Software Developer's Kit, available from Microsoft or other vendors, enabling access to ODBC data sources. Unlike using embedded SQL, no precompilation is required. Applications developed using this interface may be executed on a variety of databases without being compiled against each of the databases. Through the interface, applications use procedure calls at execution time to connect to databases, to issue SQL statements, and to get returned data and status information.

The DB2 CLI interface provides many features not available in embedded SQL. For example:

- CLI provides function calls which support a consistent way to query and retrieve database system catalog information across the DB2 family of database management systems. This reduces the need to write application server specific catalog queries.
- Stored procedures called from application programs written using CLI can return result sets to those programs.

For a complete description of all the available functions, and their syntax, see SQL Call Level Interfaces (ODBC) book.

Java DataBase Connectivity (JDBC) and embedded SQL for Java (SQLJ) programs

DB2 for i implements two standards-based Java programming APIs: Java Database Connectivity (JDBC) and embedded SQL for Java (SQLJ). Both can be used to create Java applications and applets that access DB2.

JDBC calls are translated to calls to DB2 CLI through Java native methods. You can access DB2 for i databases through two JDBC drivers: IBM Developer Kit for Java driver or IBM Toolbox for Java JDBC driver. For specific information about the IBM Toolbox for Java JDBC driver, see IBM Toolbox for Java.

Static SQL cannot be used by JDBC. SQLJ applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can also contain embedded static SQL statements in the SQLJ source files. An SQLJ source file has to be translated with the SQLJ translator before the resulting Java source code can be compiled.

For more information about JDBC and SQLJ applications, refer to the Developer Kit for Java book.

OLE DB and ADO (ActiveX Data Object)

IBM i Access for Windows includes OLE DB Providers, along with the Programmer's Toolkit to allow DB2 client/server application development quick and easy from the Windows client PC. For more information, refer to the IBM i Access for Windows OLE DB provider in the IBM i Information Center.

.NET

IBM i Access for Windows include a .NET Provider to allow DB2 client/server application development quick and easy from the Windows client PC. For more

information, refer to the System i Access for Windows .NET provider in the IBM i Information Center.

Schemas

The objects in a relational database are organized into sets called schemas. A schema provides a logical classification of objects in a relational database.

A *schema name* is used as the qualifier of SQL object names such as tables, views, indexes, and triggers. A schema is also called a collection or library.

A schema has a name and may have a different system name. The system name is the name used by the IBM i operating system. Either name is acceptable wherever a schema-name is specified in SQL statements.

A schema is distinct from, and should not be confused with, an XML schema, which is a standard that describes the structure and validates the content of XML documents.

Each database manager supports a set of schemas that are reserved for use by the database manager. Such schemas are called *system schemas*. The schema SESSION and all schemas that start with 'SYS' and 'Q' are *system schemas*.

User objects must not be created in *system schemas*, other than SESSION. SESSION is always used as the schema name for declared temporary tables. Users should not create schemas that start with 'SYS' or 'Q'.

A schema is also an object in the relational database. It is explicitly created using the CREATE SCHEMA statement.[1] For more information, see “CREATE SCHEMA” on page 968.

An object that is contained in a schema is assigned to the schema when the object is created. The schema to which it is assigned is determined by the name of the object if specifically qualified with a schema name or by the default schema name if not qualified.

For example, a user creates a schema called C:

```
CREATE SCHEMA C
```

The user can then issue the following statement to create a table called X in schema C:

```
CREATE TABLE C.X (COL1 INT)
```

Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows. There is no inherent order of the rows within a table. At the intersection of every column and row is a specific data item called a *value*. A *column* is a set of values of the same type. A *row* is a sequence of values such that the *n*th value is a value of the *n*th column of the table.

There are three types of tables:

1. A schema can also be created using the CRTLIB CL command, however, the catalog views and journal and journal receiver created by using the CREATE SCHEMA statement will not be created with CRTLIB.

- A *base table* is created with the CREATE TABLE statement and is used to hold persistent user data. For more information see "CREATE TABLE" on page 982.

  A base table has a name and may have a different system name. The system name is the name used by the IBM i operating system. Either name is acceptable wherever a *table-name* is specified in SQL statements.

  A column of a base table has a name and may have a different system column name. The system column name is the name used by the IBM i operating system. Either name is acceptable wherever *column-name* is specified in SQL statements. For more information see "CREATE TABLE" on page 982.

  A *materialized query table* is a base table created with the CREATE TABLE statement and used to contain data that is derived (materialized) from a *select-statement*. A source table is a base table, view, table expression, or user-defined table function. The *select-statement* specifies the query that is used to refresh the data in the materialized query table.

  Materialized query tables can be used to improve the performance of SQL queries. If the database manager determines that a portion of a query could be resolved by using the data in a materialized query table, the query may be rewritten by the database manager to use the materialized query table. For more information about creating materialized query tables, see "CREATE TABLE" on page 982.

  A *partitioned table* is a table whose data is contained in one or more local partitions (members). There are two mechanisms that can be specified to determine into which partition a specific row will be inserted. Range partitioning allows a user to specify different ranges of values for each partition. When a row is inserted, the values specified in the row are compared to the specified ranges to determine which partition is appropriate. Hash partitioning allows a user to specify a partitioning key on which a hash algorithm is used to determine which partition is appropriate. The *partitioning key* is a set of one or more columns in a partitioned table that are used to determine in which partition a row belongs.

  A *distributed table* is a table whose data is partitioned across a nodegroup. A *nodegroup* is an object that provides a logical grouping of a set of two or more systems. The *partitioning key* is a set of one or more columns in a distributed table that are used to determine on which system a row belongs. For more information about distributed tables, see the DB2 Multisystem book.
- A *result table* is a set of rows that the database manager selects or generates from a query. For information on queries, see Chapter 5, "Queries," on page 627.
- A *declared temporary table* is created with a DECLARE GLOBAL TEMPORARY TABLE statement and is used to hold temporary data on behalf of a single application. This table is dropped implicitly when the application disconnects from the database.

Keys

A *key* is one or more expressions that are identified as such in the description of an index, unique constraint, or a referential constraint. The same expression can be part of more than one key.

A *composite key* is an ordered set of expressions of the same base table. The ordering of the expressions is not constrained by their ordering within the base table. The term *value* when used with respect to a composite key denotes a composite value. Thus, a rule such as "the value of the foreign key must be equal

to the value of the primary key" means that each component of the value of the foreign key must be equal to the corresponding component of the value of the primary key.

Constraints

A *constraint* is a rule that the database manager enforces.

There are three types of constraints:

- A *unique constraint* is a rule that forbids duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, a unique constraint can be defined on the supplier identifier in the supplier table to ensure that the same supplier identifier is not given to two suppliers.
- A *referential constraint* is a logical rule about values in one or more columns in one or more tables. For example, a set of tables shares information about a corporation's suppliers. Occasionally, a supplier's ID changes. You can define a referential constraint stating that the ID of the supplier in a table must match a supplier ID in the supplier information. This constraint prevents insert, update, or delete operations that would otherwise result in missing supplier information.
- A *check constraint* sets restrictions on data added to a specific table. For example, a check constraint can ensure that the salary level for an employee is at least $20 000 whenever salary data is added or updated in a table containing personnel information.

Unique constraints

A *unique constraint* is the rule that the values of a key are valid only if they are unique. A key that is constrained to have unique values is called a *unique key*. A unique constraint is enforced by using a unique index. The unique index is used by the database manager to enforce the uniqueness of the values of the key during the execution of INSERT and UPDATE statements.

There are two types of unique constraints:

- Unique keys can be defined as a primary key using a CREATE TABLE or ALTER TABLE statement. A base table cannot have more than one primary key. A CHECK constraint will be added implicitly to enforce the rule that the NULL value is not allowed in the columns that make up the primary key. A unique index on a primary key is called a *primary index*.
- Unique keys can be defined using the UNIQUE clause of the CREATE TABLE or ALTER TABLE statement. A base table can have more than one set of UNIQUE keys.

A unique key that is referenced by the foreign key of a referential constraint is called the *parent key*. A parent key is either a primary key or a UNIQUE key. When a base table is defined as a parent in a referential constraint, the default parent key is its primary key.

The unique index that is used to enforce a unique constraint is implicitly created when the unique constraint is defined. Alternatively, it can be defined by using the CREATE UNIQUE INDEX statement.

For more information about defining unique constraints, see "ALTER TABLE" on page 759 or "CREATE TABLE" on page 982.

Referential constraints

Referential integrity is the state of a database in which all values of all foreign keys are valid. A *foreign key* is a key that is part of the definition of a referential constraint.

A *referential constraint* is the rule that the values of the foreign key are valid only if:

- They appear as values of a parent key, or
- Some component of the foreign key is null.

The base table containing the parent key is called the *parent table* of the referential constraint, and the base table containing the foreign key is said to be a *dependent* of that table.

Referential constraints are optional and can be defined in CREATE TABLE statements and ALTER TABLE statements. Referential constraints are enforced by the database manager during the execution of INSERT, UPDATE, and DELETE statements. The enforcement is effectively performed at the completion of the statement except for delete and update rules of RESTRICT which are enforced as rows are processed.

Referential constraints with a delete or update rule of RESTRICT are always enforced before any other referential constraints. Other referential constraints are enforced in an order independent manner. That is, the order does not affect the result of the operation. Within an SQL statement:

- A row can be marked for deletion by any number of referential constraints with a delete rule of CASCADE.
- A row can only be updated by one referential constraint with a delete rule of SET NULL or SET DEFAULT.
- A row that was updated by a referential constraint cannot also be marked for deletion by another referential constraint with a delete rule of CASCADE.

The rules of referential integrity involve the following concepts and terminology:

Parent key
A primary key or unique key of a referential constraint.

Parent row
A row that has at least one dependent row.

Parent table
A base table that is a parent in at least one referential constraint. A base table can be defined as a parent in an arbitrary number of referential constraints.

Dependent table
A base table that is a dependent in at least one referential constraint. A base table can be defined as a dependent in an arbitrary number of referential constraints. A dependent table can also be a parent table.

Descendent table
A base table is a descendent of base table T if it is a dependent of T or a descendent of a dependent of T.

Dependent row
A row that has at least one parent row.

Descendent row
A row is a descendent of row p if it is a dependent of p or a descendent of a dependent of p.

Referential cycle
A set of referential constraints such that each table in the set is a descendent of itself.

Self-referencing row
A row that is a parent of itself.

Self-referencing table
A base table that is a parent and a dependent in the same referential constraint. The constraint is called a *self-referencing constraint*.

The insert rule of a referential constraint is that a nonnull insert value of the foreign key must match some value of the parent key of the parent table. The value of a composite foreign key is null if any component of the value is null.

The update rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION and RESTRICT. The update rule applies when a row of the parent or dependent table is updated. The update rule of a referential constraint is that a nonnull update value of a foreign key must match some value of the parent key of the parent table. The value of a composite foreign key is treated as null if any component of the value is null.

The delete rule of a referential constraint is specified when the referential constraint is defined. The choices are RESTRICT, NO ACTION, CASCADE, SET NULL or SET DEFAULT. SET NULL can be specified only if some column of the foreign key allows null values.

The delete rule of a referential constraint applies when a row of the parent table is deleted. More precisely, the rule applies when a row of the parent table is the object of a delete or propagated delete operation (defined below) and that row has dependents in the dependent table of the referential constraint. Let P denote the parent table, let D denote the dependent table, and let p denote a parent row that is the object of a delete or propagated delete operation. If the delete rule is:

- RESTRICT or NO ACTION, an error is returned and no rows are deleted
- CASCADE, the delete operation is propagated to the dependents of p in D
- SET NULL, each nullable column of the foreign key of each dependent of p in D is set to null
- SET DEFAULT, each column of the foreign key of each dependent of p in D is set to its default value

Each referential constraint in which a table is a parent has its own delete rule, and all applicable delete rules are used to determine the result of a delete operation. Thus, a row cannot be deleted if it has dependents in a referential constraint with a delete rule of RESTRICT or NO ACTION, or if the deletion cascades to any of its descendants that are dependents in a referential constraint with the delete rule of RESTRICT or NO ACTION.

The deletion of a row from parent table P involves other tables and may affect rows of these tables:

- If table D is a dependent of P and the delete rule is RESTRICT or NO ACTION, D is involved in the operation but is not affected by the operation.

- If D is a dependent of P and the delete rule is SET NULL, D is involved in the operation, and rows of D may be updated during the operation.
- If D is a dependent of P and the delete rule is SET DEFAULT, D is involved in the operation, and rows of D may be updated during the operation.
- If D is a dependent of P and the delete rule is CASCADE, D is involved in the operation and rows of D may be deleted during the operation.

  If rows of D are deleted, the delete operation on P is said to be propagated to D. If D is also a parent table, the actions described in this list apply, in turn, to the dependents of D.

Any base table that may be involved in a delete operation on P is said to be *delete-connected* to P. Thus, a base table is delete-connected to base table P if it is a dependent of P or a dependent of a base table to which delete operations from P cascade.

For more information on defining referential constraints, see "ALTER TABLE" on page 759 or "CREATE TABLE" on page 982.

Check constraints

A *check constraint* is a rule that specifies which values are allowed in every row of a base table. The definition of a check constraint contains a search condition that must not be FALSE for any row of the base table.

Each column referenced in the search condition of a check constraint on a table T must identify a column of T. For more information about search conditions, see "Search conditions" on page 225.

A base table can have more than one check constraint. Each check constraint defined on a base table is enforced by the database manager when either of the following occur:

- A row is inserted into that base table.
- A row of that base table is updated.

A check constraint is enforced by applying its search condition to each row that is inserted or updated in that base table. An error is returned if the result of the search condition is FALSE for any row.

For more information about defining check constraints, see "ALTER TABLE" on page 759 or "CREATE TABLE" on page 982.

Indexes

An *index* is a set of pointers to rows of a base table. Each index is based on the values of data in one or more table columns. An index is an object that is separate from the data in the table. When an index is created, the database manager builds this structure and maintains it automatically.

An index has a name and may have a different system name. The system name is the name used by the IBM i operating system. Either name is acceptable wherever an *index-name* is specified in SQL statements. For more information, see "CREATE INDEX" on page 929.

The database manager uses two types of indexes:

- Binary radix tree index

Binary radix tree indexes provide a specific order to the rows of a table. The database manager uses them to:

- Improve performance. In most cases, access to data is faster than without an index.
- Ensure uniqueness. A table with a unique index cannot have rows with identical keys.

- Encoded vector index

  Encoded vector indexes do not provide a specific order to the rows of a table. The database manager only uses these indexes to improve performance.

  An encoded vector access path works with the help of encoded vector indexes and provides access to a database file by assigning codes to distinct key values and then representing these values in an array. The elements of the array can be 1, 2, or 4 bytes in length, depending on the number of distinct values that must be represented. Because of their compact size and relative simplicity, encoded vector access paths provide for faster scans that can be more easily processed in parallel.

An *index* is created with the CREATE INDEX statement. For more information about creating indexes, see "CREATE INDEX" on page 929.

For more information about indexes, see IBM DB2 for i indexing methods and strategies  .

Triggers

A *trigger* defines a set of actions that are executed automatically whenever a delete, insert, or update operation occurs on a specified table or view. When such an SQL operation is executed, the trigger is said to be activated.

The set of actions can include almost any operation allowed on the system. A few operations are not allowed, such as:

- Commit or rollback (if the same commitment definition is used for the trigger actions and the triggering event)
- CONNECT, SET CONNECTION, DISCONNECT, and RELEASE statements
- SET SESSION AUTHORIZATION

For a complete list of restrictions, see "CREATE TRIGGER" on page 1033 and the Database Programming book.

Triggers can be used along with referential constraints and check constraints to enforce data integrity rules. Triggers are more powerful than constraints because they can also be used to cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions that perform operations both inside and outside of database manager. For example, instead of preventing an update to a column if the new value exceeds a certain amount, a trigger can substitute a valid value and send a notice to an administrator about the invalid update.

Triggers are a useful mechanism to define and enforce transitional business rules that involve different states of the data (for example, salary cannot be increased by more than 10 percent). Such a limit requires comparing the value of a salary before and after an increase. For rules that do not involve more than one state of the data, consider using referential and check constraints.

Triggers also move the application logic that is required to enforce business rules into the database, which can result in faster application development and easier maintenance because the business rule is no longer repeated in several applications, but one version is centralized to the trigger. With the logic in the database, for example, the previously mentioned limit on increases to the salary column of a table, database manager checks the validity of the changes that any application makes to the salary column. In addition, the application programs do not need to be changed when the logic changes.

For more information about creating triggers, see "CREATE TRIGGER" on page 1033. [2]

Triggers are optional and are defined using the CREATE TRIGGER statement or the **Add Physical File Trigger (ADDPFTRG)** CL command. Triggers are dropped using the DROP TRIGGER statement or the **Remove Physical File Trigger (RMVPFTRG)** CL command. For more information about creating triggers, see the CREATE TRIGGER statement. For more information about triggers in general, see the "CREATE TRIGGER" on page 1033 statement or the SQL Programming and the Database Programming books.

There are a number of criteria that are defined when creating a trigger which are used to determine when a trigger should be activated.

- The *subject table* defines the table or view for which the trigger is defined.
- The *trigger event* defines a specific SQL operation that modifies the subject table. The operation could be delete, insert, or update.
- The *trigger activation time* defines whether the trigger should be activated before or after the trigger event is performed on the subject table.

The statement that causes a trigger to be activated will include a *set of affected rows*. These are the rows of the subject table that are being deleted, inserted or updated. The *trigger granularity* defines whether the actions of the trigger will be performed once for the statement or once for each of the rows in the set of affected rows.

The *trigger action* consists of an optional search condition and a set of SQL statements that are executed whenever the trigger is activated. The SQL statements are only executed if no search condition is specified or the specified search condition evaluates to true.

The triggered action may refer to the values in the set of affected rows. This is supported through the use of *transition variables*. Transition variables use the names of the columns in the subject table qualified by a specified name that identifies whether the reference is to the old value (prior to the update) or the new value (after the update). The new value can also be changed using the SET transition-variable statement in before update or insert triggers. Another means of referring to the values in the set of affected rows is using *transition tables*. Transition tables also use the names of the columns of the subject table but have a name specified that allows the complete set of affected rows to be treated as a table. Transition tables can only be used in after triggers. Separate transition tables can be defined for old and new values.

2. The ADDPFTRG CL command also defines a trigger, including triggers that are activated on any read operation.

Multiple triggers can be specified for a combination of table, event, or activation time. The order in which the triggers are activated is the same as the order in which they were created. Thus, the most recently created trigger will be the last trigger activated.

The activation of a trigger may cause *trigger cascading*. This is the result of the activation of one trigger that executes SQL statements that cause the activation of other triggers or even the same trigger again. The triggered actions may also cause updates as a result of the original modification, which may result in the activation of additional triggers. With trigger cascading, a significant chain of triggers may be activated causing significant change to the database as a result of a single delete, insert or update statement.

The actions performed in the trigger are considered to be part of the operation that caused the trigger to be executed. Thus, when the isolation level is anything other than NC (No Commit) and the trigger actions are performed using the same commitment definition as the trigger event:

- The database manager ensures that the operation and the triggers executed as a result of that operation either all complete or are backed out. Operations that occurred prior to the triggering operation are not affected.
- The database manager effectively checks all constraints (except for a constraint with a RESTRICT delete rule) after the operation and the associated triggers have been executed.

A trigger has an attribute that specifies whether it is allowed to delete or update a row that has already been inserted or updated within the SQL statement that caused the trigger to be executed.

- If ALWREPCHG(*YES) is specified when the trigger is defined, then within an SQL statement:
  - The trigger is allowed to update or delete any row that was inserted or already updated by that same SQL statement. This also includes any rows inserted or updated by a trigger or referential constraint caused by the same SQL statement.
- If ALWREPCHG(*NO) is specified when the trigger is defined, then within an SQL statement:
  - A row can be deleted by a trigger only if that row has not been inserted or updated by that same SQL statement. If the isolation level is anything other than NC (No Commit) and the trigger actions are performed using the same commitment definition as the trigger event, this also includes any inserts or updates by a trigger or referential constraint caused by the same SQL statement.
  - A row can be updated by a trigger only if that row has not already been inserted or updated by that same SQL statement. If the isolation level is anything other than NC (No Commit) and the trigger actions are performed using the same commitment definition as the trigger event, this also includes any inserts or updates by a trigger or referential constraint caused by the same SQL statement.

All triggers created by using the CREATE TRIGGER statement implicitly have the ALWREPCHG(*YES) attribute.

Views

A *view* provides an alternative way of looking at the data in one or more tables.

Views

A view is a named specification of a result table. The specification is a SELECT statement that is effectively executed whenever the view is referenced in an SQL statement. Thus, a view can be thought of as having columns and rows just like a base table. For retrieval, all views can be used just like base tables. Whether a view can be used in an insert, update, or delete operation depends on its definition.

An index cannot be created for a view. However, an index created for a table on which a view is based may improve the performance of operations on the view.

When the column of a view is directly derived from a column of a base table, that column inherits any constraints that apply to the column of the base table. For example, if a view includes a foreign key of its base table, INSERT and UPDATE operations using that view are subject to the same referential constraints as the base table. Likewise, if the base table of a view is a parent table, DELETE operations using that view are subject to the same rules as DELETE operations on the base table. A view also inherits any triggers that apply to its base table. For example, if the base table of a view has an update trigger, the trigger is fired when an update is performed on the view.

A view has a name and may have a different system name. The system name is the name used by the IBM i operating system. Either name is acceptable wherever a *view-name* is specified in SQL statements.

A column of a view has a name and may have a different system column name. The system column name is the name used by the IBM i operating system. Either name is acceptable wherever *column-name* is specified in SQL statements.

A *view* is created with the CREATE VIEW statement. For more information about creating views, see "CREATE VIEW" on page 1069.

User-defined types

A *user-defined type* is a data type that is defined to the database using a CREATE statement.

There are two types of user-defined data type:

- Distinct type
- Array type

A *distinct type* is a user-defined type that shares its internal representation with a built-in data type (its source type), but is considered to be a separate and incompatible data type for most operations. A distinct type is created with an SQL CREATE TYPE (Distinct) statement. A distinct type can be used to define a column of a table, or a parameter of a routine. For more information, see "CREATE TYPE (Distinct)" on page 1055 and "User-defined types" on page 90.

An *array type* is a user-defined type that defines a one column array of a built-in data type. An array type is created with an SQL CREATE TYPE (Array) statement. An array type can be used as a parameter of a procedure and as a variable in an SQL procedure. For more information, see "CREATE TYPE (Array)" on page 1050 and "User-defined types" on page 90.

Aliases

An *alias* is an alternate name for a table or view.

An alias can be used to reference a table or view in cases where an existing table or view can be referenced.[3] However, the option of referencing a table or view by an alias is not explicitly shown in the syntax diagrams or mentioned in the description of SQL statements. Like tables and views, an alias may be created, dropped, and have a comment or label associated with it. No authority is necessary to use an alias. Access to the tables and views that are referred to by the alias, however, still requires the appropriate authorization for the current statement.

An alias has a name and may have a different system name. The system name is the name used by the IBM i operating system. Either name is acceptable wherever an *alias-name* is specified in SQL statements.

An *alias* is created with the CREATE ALIAS statement. For more information about creating aliases, see "CREATE ALIAS" on page 847.

Packages and access plans

A *package* is an object that contains control structures used to execute SQL statements.

Packages are produced during distributed program preparation. The control structures can be thought of as the bound or operational form of SQL statements.[4] All control structures in a package are derived from the SQL statements embedded in a single source program.

In this book, the term *access plan* is used in general for packages, procedures, functions, triggers, and programs or service programs that contain control structures used to execute SQL statements. For example, the description of the DROP statement says that dropping an object also invalidates any access plans that reference the object (see "DROP" on page 1151). This means that any packages, procedures, functions, triggers, and programs or service programs containing control structures referencing the dropped object are invalidated.

An invalidated *access plan* will be implicitly rebuilt the next time its associated SQL statement is executed. For example, if an index is dropped that is used in an *access plan* for a SELECT INTO statement, the next time that SELECT INTO statement is executed, the access plan will be rebuilt.

A package can also be created by the Process Extended Dynamic SQL (QSQPRCED) API. Packages created by the Process Extended Dynamic SQL (QSQPRCED) API can only be used by the Process Extended Dynamic SQL (QSQPRCED) API. They cannot be used at an application server through DRDA® protocols. For more information, see the Database and File APIs information in the **Programming** category of the IBM i Information Center.

The QSQPRCED API is used by IBM i Access for Windows to create packages for caching SQL statements executed via ODBC, JDBC, SQLJ, OLD DB, and .NET interfaces.

3. You cannot use an alias in all contexts. For example, an alias that refers to an individual member of a database file cannot be used in most SQL schema statements. For more information, see "CREATE ALIAS" on page 847.

4. For non-distributed SQL programs, non-distributed service programs, SQL functions, and SQL procedures, the control structures used to execute SQL statements are stored in the associated space of the object.

Routines

A *routine* is an executable SQL object.

There are two types of routines.

Functions

A *function* is a routine that can be invoked from within other SQL statements and returns a value or a table. For more information, see "Functions" on page 158.

Functions are classified as either SQL functions or external functions. SQL functions are written using SQL statements, which are also known collectively as SQL procedural language. External functions reference a host language program which may or may not contain SQL statements.

A *function* is created with the CREATE FUNCTION statement. For more information about creating functions, see "CREATE FUNCTION" on page 851.

Procedures

A *procedure* (sometimes called a *stored procedure*) is a routine that can be called to perform operations that can include both host language statements and SQL statements.

Procedures are classified as either SQL procedures or external procedures. SQL procedures are written using SQL statements, which are also known collectively as SQL procedural language. External procedures reference a host language program which may or may not contain SQL statements.

A *procedure* is created with the CREATE PROCEDURE statement. For more information about creating procedures, see "CREATE PROCEDURE" on page 937.

Procedures in SQL provide the same benefits as procedures in a host language. That is, a common piece of code need only be written and maintained once and can be called from several programs. Both host languages and SQL can call procedures that exist on the local system. However, SQL can also call a procedure that exists on a remote system. In fact, the major benefit of procedures in SQL is that they can be used to enhance the performance characteristics of distributed applications.

Assume that several SQL statements must be executed at a remote system. There are two ways this can be done. Without procedures, when the first SQL statement is executed, the application requester will send a request to an application server to perform the operation. It then waits for a reply that indicates whether the statement is executed successfully or not and optionally returns results. When the second and each subsequent SQL statement is executed, the application requester will send another request and wait for another reply.

If the same SQL statements are stored in a procedure at an application server, a CALL statement can be executed that references the remote procedure. When the CALL statement is executed, the application requester will send a single request to the current server to call the procedure. It will then wait for a single reply that indicates whether the procedure executed successfully or not and optionally returns results.

The following two figures illustrate the way stored procedures can be used in a distributed application to eliminate some of the remote requests. Figure 1 shows a program making many remote requests.

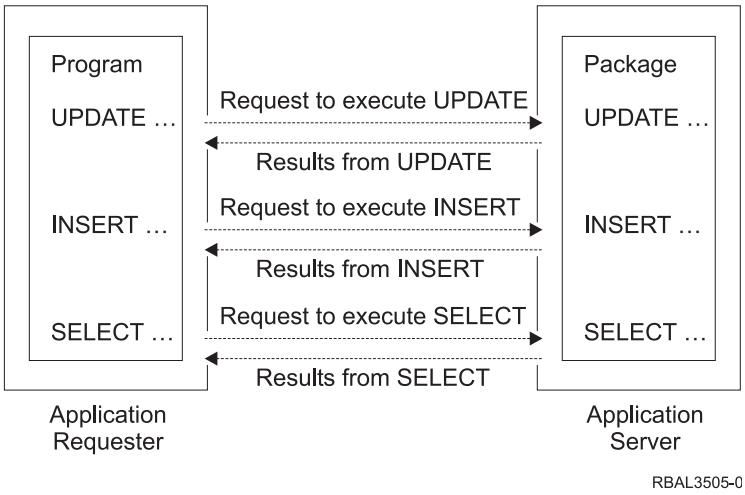


Figure 1. Application Without Remote Procedure

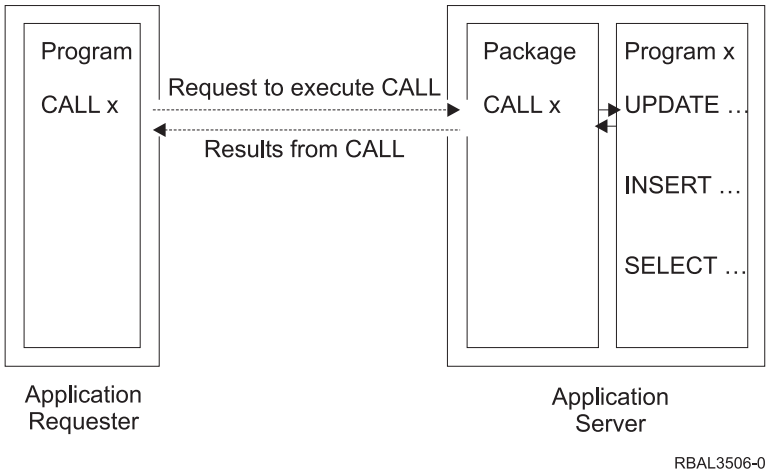


Figure 2. Application With Remote Procedure

Sequences

A sequence is a stored object that simply generates a sequence of numbers in a monotonically ascending (or descending) order. Sequences provide a way to have the database manager automatically generate unique integer and decimal primary keys, and to coordinate keys across multiple rows and tables.

A sequence can be used to exploit parallelization, instead of programmatically generating unique numbers by locking the most recently used value and then incrementing it.

Sequences are ideally suited to the task of generating unique key values. One sequence can be used for many tables, or a separate sequence can be created for each table requiring generated keys. A sequence has the following properties:

- Can have guaranteed, unique values, assuming that the sequence is not reset and does not allow the values to cycle.
- Can have increasing or decreasing values within a defined range.
- Can have an increment value other than 1 between consecutive values (the default is 1).
- Is recoverable.

Values for a given sequence are automatically generated by the database manager. Use of a sequence in the database avoids the performance bottleneck that results when an application implements sequences outside the database. The counter for the sequence is incremented (or decremented) independently from the transaction.

In some cases, gaps can be introduced in a sequence. A gap can occur when a given transaction increments a sequence two times. The transaction may see a gap in the two numbers that are generated because there may be other transactions concurrently incrementing the same sequence. A user may not realize that other users are drawing from the same sequence. Furthermore, it is possible that a given sequence can appear to have generated gaps in the numbers, because a transaction that may have generated a sequence number may have rolled back. Updating a sequence is not part of a transaction's unit of recovery.

A sequence is created with a CREATE SEQUENCE statement. A sequence can be referenced using a *sequence-reference*. A sequence reference can appear most places that an expression can appear. A sequence reference can specify whether the value to be returned is a newly generated value, or the previously generated value. For more information, see “Sequence reference” on page 195 and “CREATE SEQUENCE” on page 974.

Although there are similarities, a sequence is different than an identity column. A sequence is an object, whereas an identity column is a part of a table. A sequence can be used with multiple tables, but an identity column is part of a single table.

Authorization, privileges and object ownership

Users (identified by an authorization ID) can successfully execute SQL statements only if they have the authority to perform the specified function. To create a table, a user must be authorized to create tables; to alter a table, a user must be authorized to alter the table; and so forth.

There are two forms of authorization:

administrative authority
The person or persons holding *administrative authority* are charged with the task of controlling the database manager and are responsible for the safety and integrity of the data. Those with *administrative authority* implicitly have all privileges on all objects and control who will have access to the database manager and the extent of this access.

The security officer and all users with *ALLOBJ authority have administrative authority.

privileges
Privileges are those activities that a user is allowed to perform. Authorized users can create objects, have access to objects they own, and can pass on *privileges* on their own objects to other users by using the GRANT statement.

Privileges may be granted to specific users or to PUBLIC. PUBLIC specifies that a privilege is granted to a set of users (authorization IDs). The set consists of those users (including future users) that do not have privately granted privileges on the table or view. This affects private grants. For example, if SELECT has been granted to PUBLIC, and UPDATE is then granted to HERNANDZ, this private grant prevents HERNANDZ from having the SELECT privilege.

The REVOKE statement can be used to REVOKE previously granted *privileges*. A revoke of a privilege from an authorization ID revokes the privilege granted by all authorization IDs. Revoking a privilege from an authorization ID will not revoke that same privilege from any other authorization IDs that were granted the privilege by that authorization ID.

When an object is created, the authorization ID of the statement must have the privilege to create objects in the implicitly or explicitly specified schema. The authorization ID of a statement has the privilege to create objects in the schema if:

- it is the owner of the schema, or
- it has *OBJOPR, *EXECUTE, and *ADD to the schema.

When an object is created, one authorization ID is assigned *ownership* of the object. Ownership gives the user complete control over the object, including the privilege to drop the object. The privileges on the object can be granted by the owner, and can be revoked from the owner. In this case, the owner may temporarily be unable to perform an operation that requires that privilege. Because he is the owner, however, he is always allowed to grant the privilege back to himself.

When an object is created:

- If SQL names were specified, the *owner* of the object is the user profile with the same name as the schema into which the object is created, if a user profile with that name exists. Otherwise, the *owner* of the object is the user profile or group user profile of the job executing the statement.
- If system names were specified, the *owner* of the object is the user profile or group user profile of the job executing the statement.

Authority granted to *PUBLIC on SQL objects depends on the naming convention that is used at the time of object creation. If *SYS naming convention is used, *PUBLIC acquires the create authority (CRTAUT) of the library into which the object was created. If *SQL naming convention is used, *PUBLIC acquires *EXCLUDE authority.

In the Authorization sections of this book, it is assumed that the owner of an object has not had any privileges revoked from that object since it was initially created. If the object is a view, it is also assumed that the owner of the view has not had the system authority *READ revoked from any of the tables or views that this view is directly or indirectly dependent on. The owner has system authority *READ for all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth. For more information about authority and privileges, see Security Reference.

Catalog

The database manager maintains a set of tables containing information about objects in the database. These tables and views are collectively known as the *catalog*. The *catalog tables* contain information about objects such as tables, views, indexes, packages, and constraints.

Tables and views in the catalog are similar to any other database tables and views. Any user that has the SELECT privilege on a catalog table or view can read the data in the catalog table or view. A user cannot directly modify a catalog table or view, however. The database manager ensures that the catalog contains accurate descriptions of the objects in the database at all times.

The database manager provides a set of views that provide more consistency with the catalog views of other IBM SQL products and another set of catalog views that provide compatibility with the catalog views of the ANSI and ISO standard (called *Information Schema* in the standard).

If a schema is created using the CREATE SCHEMA statement, the schema will also contain a set of views that only contain information about objects in the schema.

For more information about catalog tables and views, see Appendix F, "DB2 for i catalog views," on page 1557.

Application processes, concurrency, and recovery

All SQL programs execute as part of an *application process*. In the IBM i operating system, an application process is called a job. In the case of ODBC, JDBC, OLE DB, .NET, and DRDA, the application process ends when the connection ends even though the job they are using does not end and may be reused.

An application process is made up of one or more activation groups. Each activation group involves the execution of one or more programs. Programs run under a non-default activation group or the default activation group. All programs except those created by ILE compilers run under the default activation group. For example, LANGUAGE JAVA external functions run under the default activation group.

For more information about activation groups, see the book ILE Concepts  .

An application process that uses commitment control can run with one or more commitment definitions. A commitment definition provides a means to scope commitment control at an activation group level or at a job level. At any given time, an activation group that uses commitment control is associated with only one of the commitment definitions.

A commitment definition can be explicitly started through the Start Commitment Control (STRCMTCTL) command. If not already started, a commitment definition is implicitly started when the first SQL statement is executed under an isolation level different than COMMIT(*NONE). More than one activation group can share a job commitment definition.

Figure 3 on page 21 shows the relationship of an application process, the activation groups in that application process, and the commitment definitions. Activation groups A and B run with commitment control scoped to the activation group.

These activation groups have their own commitment definitions. Activation group C does not run with any commitment control and does not have a commitment definition.

Application Process
Without Job-Level Commitment Definition

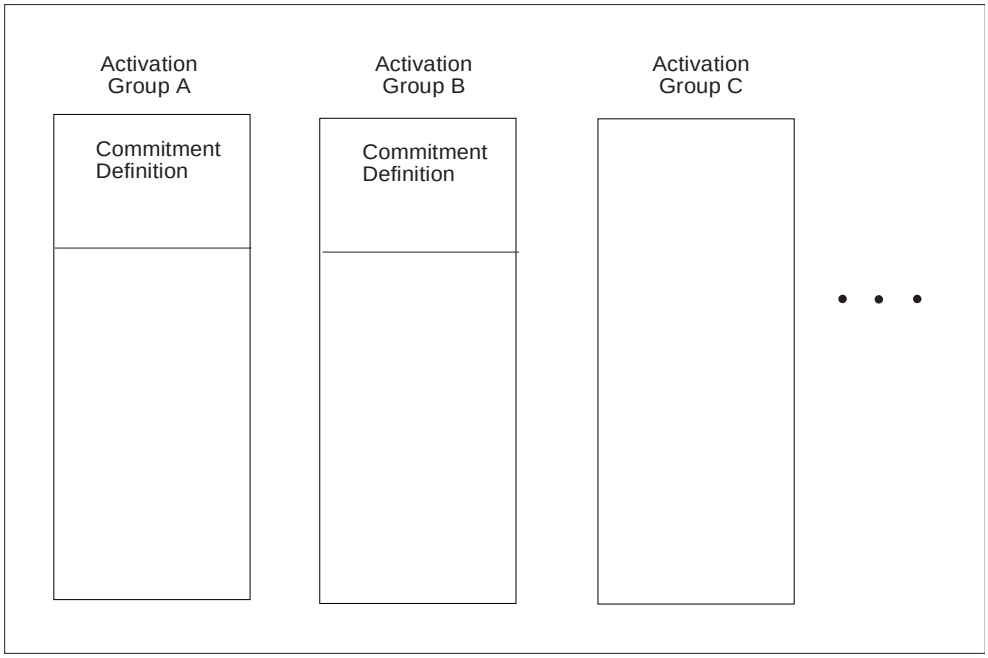


Figure 3. Activation Groups without Job Commitment Definition

Figure 4 on page 22 shows an application process, the activation groups in that application process, and the commitment definitions. Some of the activation groups are running with the job commitment definition. Activation groups A and B are running under the job commitment definition. Any commit or rollback operation in activation group A or B affects both because the commitment control is scoped to the same commitment definition. Activation group C in this example has a separate commitment definition. Commit and rollback operations performed in this activation group only affect operations within C.

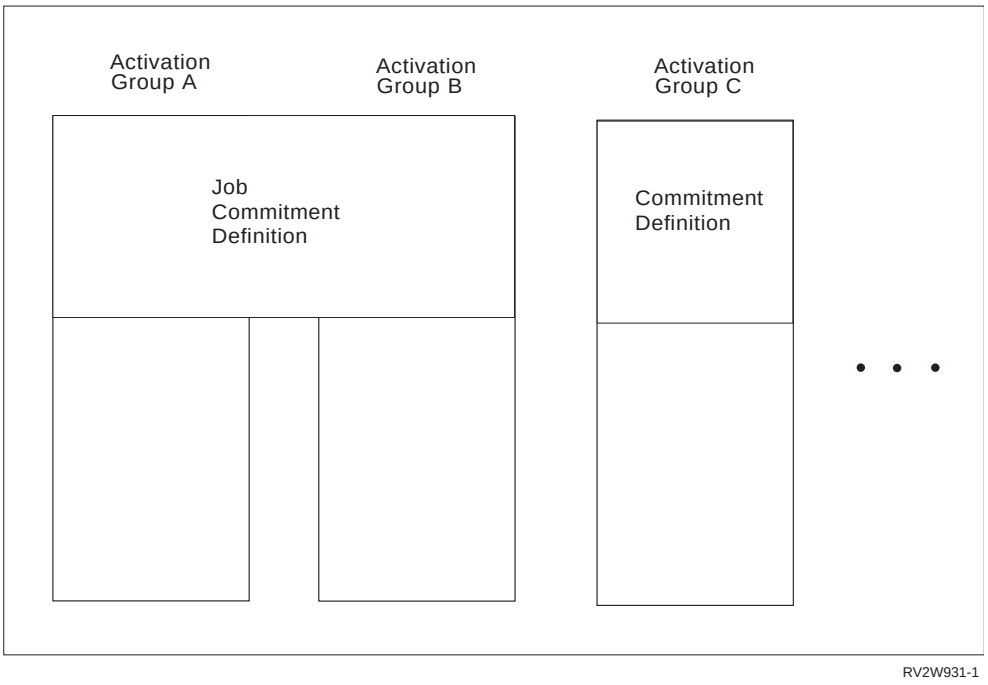


Figure 4. Activation Groups with Job Commitment Definition

For more information about commitment definitions, see the Commitment control topic.

Locking, commit, and rollback

Application processes and activation groups that use different commitment definitions can request access to the same data at the same time. *Locking* is used to maintain data integrity under such conditions. Locking prevents such things as two application processes updating the same row of data simultaneously.

The database manager acquires locks to keep the uncommitted changes of one activation group undetected by activation groups that use a different commitment definition. Object locks and other resources are allocated to an activation group. Row locks are allocated to a commitment definition.

When an activation group other than the default activation group ends *normally*, the database manager releases all locks obtained by the activation group. A user can also explicitly request that most locks be released sooner. This operation is called *commit*. Object locks associated with cursors that remain open after commit are not released.

The recovery functions of the database manager provide a means of backing out of uncommitted changes made in a commitment definition. The database manager may implicitly back out uncommitted changes under the following situations:

- When the application process ends, all changes performed under the commitment definition associated with the default activation group are backed out. When an activation group other than the default activation group ends *abnormally*, all changes performed under the commitment definition associated with that activation group are backed out.

- When using Distributed Unit of Work and a failure occurs while attempting to commit changes on a remote system, all changes performed under the commitment definition associated with remote connection are backed out.
- When using Distributed Unit of Work and a request to back out is received from a remote system because of a failure at that site, all changes performed under the commitment definition associated with remote connection are backed out.

A user can also explicitly request that their database changes be backed out. This operation is called *rollback*.

Locks acquired by the database manager on behalf of an activation group are held until the unit of work is ended. A lock explicitly acquired by a LOCK TABLE statement can be held past the end of a unit of work if COMMIT HOLD or ROLLBACK HOLD is used to end the unit of work.

A cursor can implicitly lock the row at which the cursor is positioned. This lock prevents:

- Other cursors associated with a different commitment definition from locking the same row.
- A DELETE or UPDATE statement associated with a different commitment definition from locking the same row.

Unit of work

A *unit of work* (also known as a *transaction*, *logical unit of work*, or *unit of recovery*) is a recoverable sequence of operations. Each commitment definition involves the execution of one or more units of work. At any given time, a commitment definition has a single unit of work.

A unit of work is started either when the commitment definition is started, or when the previous unit of work is ended by a commit or rollback operation. A unit of work is ended by a commit operation, a rollback operation, or the ending of the activation group. A commit or rollback operation affects only the database changes made within the unit of work that the commit or rollback ends. While changes remain uncommitted, other activation groups using different commitment definitions running under isolation levels COMMIT(*CS), COMMIT(*RS), and COMMIT(*RR) cannot perceive the changes. The changes can be backed out until they are committed. Once changes are committed, other activation groups running in different commitment definitions can access them, and the changes can no longer be backed out.

The start and end of a unit of work defines points of consistency within an activation group. For example, a banking transaction might involve the transfer of funds from one account to another. Such a transaction would require that these funds be subtracted from the first account, and added to the second. Following the subtraction step, the data is inconsistent. Only after the funds are added to the second account is consistency established again. When both steps are complete, the commit operation can be used to end the unit of work. After the commit operation, the changes are available to activation groups that use different commitment definitions.

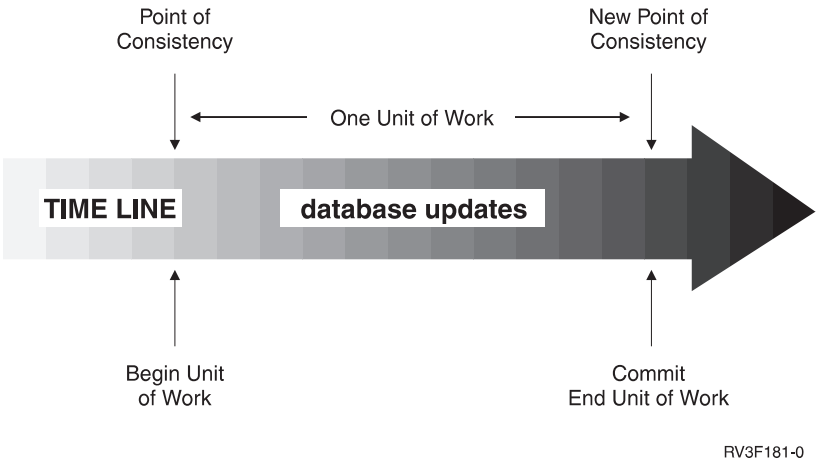


Figure 5. Unit of Work with a Commit Statement

Rolling back work

The database manager can back out all changes made in a unit of work or only selected changes. Only backing out all changes results in a point of consistency.

Rolling back all changes

The SQL ROLLBACK statement without the TO SAVEPOINT clause causes a full rollback operation. If such a rollback operation is successfully executed, database manager backs out uncommitted changes to restore the data consistency that it assumes existed when the unit of work was initiated. That is, the database manager undoes the work, as shown in the diagram below:

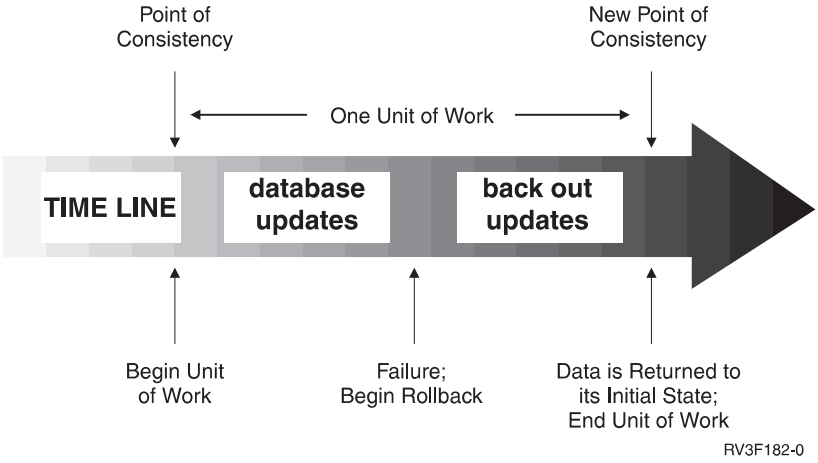


Figure 6. Unit of Work with a Rollback Statement

Rolling back selected changes using savepoints

A *savepoint* represents the state of data at some particular time during a unit of work. An application process can set savepoints within a unit of work, and then as logic dictates, roll back only the changes that were made after a savepoint was set. For example, part of a reservation transaction might involve booking an airline flight and then a hotel room. If a flight gets reserved but a hotel room cannot be

reserved, the application process might want to undo the flight reservation without undoing any database changes made in the transaction prior to making the flight reservation. SQL programs can use the SQL SAVEPOINT statement to set savepoints, the SQL ROLLBACK statement with the TO SAVEPOINT clause to undo changes to a specific savepoint or the last savepoint that was set, and the RELEASE SAVEPOINT statement to delete a savepoint.

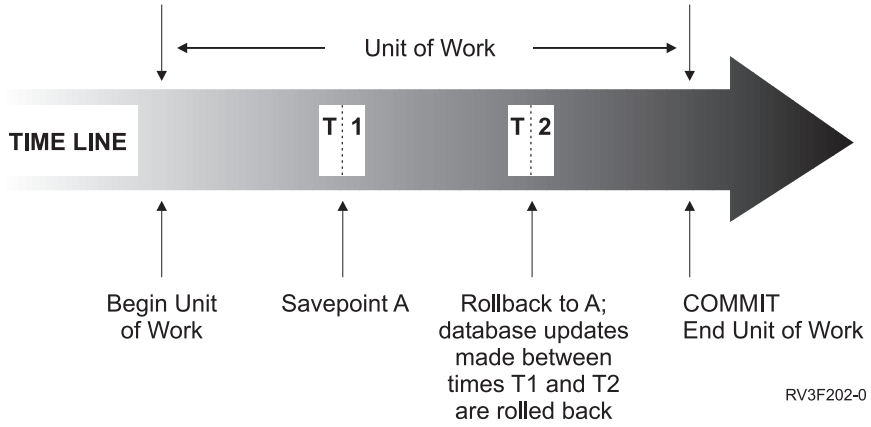


Figure 7. Unit of Work with a Rollback Statement and a Savepoint Statement

Threads

In the IBM i operating system, an application process can also consist of one or more threads. By default, a thread shares the same commitment definitions and locks as the other threads in the job. Thus, each thread can operate on the same unit of work so that when one thread commits or rolls back, it can commit or rollback all changes performed by all threads. This type of processing is useful if multiple threads are cooperating to perform a single task in parallel.

In other cases, it is useful for a thread to perform changes independent from other threads in the job. In this case, the thread would not want to share commitment definitions or lock with the other threads. Furthermore, a job can use SQL server mode in order to take more fine grain control of multiple database connections and transaction information. A typical multi-threaded job may require this control. There are several ways to accomplish this type of processing:

- Make sure the programs running in the thread use a separate activation group (be careful not to use ACTGRP(*NEW)).
- Make sure that the job is running in SQL server mode before issuing the first SQL statement. SQL server mode can be activated for a job by using one of the following mechanisms before data access occurs in the application:
  - Use the ODBC API, SQLSetEnvAttr() and set the SQL_ATTR_SERVER_MODE attribute to SQL_TRUE before doing any data access.
  - Use the Change Job API, QWTCHGJB(), and set the 'Server mode for Structured Query Language' key before doing any data access.
  - Use JAVA to access the database via JDBC. JDBC automatically uses server mode to preserve required semantics of JDBC.

When SQL server mode is established, all SQL statements are passed to an independent server job that will handle the requests. Server mode behavior for SQL behavior includes:

- For embedded SQL, each thread in a job implicitly gets one and only one connection to the database (and thus its own commitable transaction).

- For ODBC/CLI, JDBC, OLE DB, and .NET, each connection represents a stand-alone connection to the database and can be committed and used as a separate entity.

For more information, see SQL Call Level Interface (ODBC).

The following SQL support is not threadsafe:

- Remote access through DRDA
- ALTER FUNCTION
- ALTER PROCEDURE
- ALTER SEQUENCE
- ALTER TABLE
- COMMENT
- CREATE ALIAS
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE TYPE
- CREATE VARIABLE
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DROP
- GRANT
- LABEL
- REFRESH TABLE
- RENAME
- REVOKE

For more information, see Multithreaded applications in the Programming topic of the IBM i Information Center.

Isolation level

The *isolation level* used during the execution of SQL statements determines the degree to which the activation group is isolated from concurrently executing activation groups.

Thus, when activation group P executes an SQL statement, the isolation level determines:

- The degree to which rows retrieved by P and database changes made by P are available to other concurrently executing activation groups.
- The degree to which database changes made by concurrently executing activation groups can affect P.

The isolation level can be explicitly specified on a DELETE, INSERT, SELECT INTO, UPDATE, or select-statement. If the isolation level is not explicitly specified, the isolation level used when the SQL statement is executed is the *default isolation level*.

DB2 for i provides several ways to specify the *default isolation level*:

Table 2. Default Isolation Level Interfaces

SQL Interface	Specification
Embedded SQL	COMMIT parameter on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to set the COMMIT values. (For more information about CRTSQLxxx commands, see Embedded SQL Programming.)
SQL functions and procedures	Static SQL statements in SQL functions and procedures use the isolation level that was in effect at the time the SQL function or procedure was created. The SET OPTION statement (COMMIT) can be used to set the isolation level.
Run SQL Statements	COMMIT parameter on the Run SQL Statements (RUNSQLSTM) command. (For more information about the RUNSQLSTM command, see SQL Programming.)
SET TRANSACTION SQL statement	Overrides the default isolation level within a unit of work. When the unit of work ends, the isolation level returns to the value it had at the beginning of the unit of work. This statement overrides any other specification of isolation level for static and dynamic SQL statements in the unit of work. (For more information about the SET TRANSACTION statement, see "SET TRANSACTION" on page 1399.)
isolation-clause	The *isolation-clause* on the SELECT, SELECT INTO, INSERT, UPDATE, DELETE, and DECLARE CURSOR statements overrides the default isolation level for a specific statement or cursor. The isolation level is in effect only for the execution of the statement containing the *isolation-clause* and has no effect on any pending changes in the current unit of work. (For more information about the *isolation-clause*, see "isolation-clause" on page 693.)
Call Level Interface (CLI) on the server	SQL_ATTR_COMMIT or SQL_TXN_ISOLATION environment variable or connection options (For more information about CLI, see SQL Call Level Interfaces (ODBC).)
JDBC or SQLJ on the server using IBM Developer Kit for Java	transaction isolation property object (For more information about JDBC and SQLJ, see IBM Developer Kit for Java.)
ODBC on a client using the IBM i Access Family ODBC Driver	Commit Mode in ODBC Setup (For more information about ODBC, see System i Access.)

Table 2. Default Isolation Level Interfaces (continued)

SQL Interface	Specification
JDBC on a client using the IBM Toolbox for Java	Isolation Level in JDBC Setup (For more information about JDBC, see System i Access.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java.)
OLE DB on a client using the IBM i Access Family OLE DB Provider	IsolationLevel Connection Object Property (For more information about OLE DB, see System i Access.)
ADO .NET on a client using the IBM i Access Family ADO .NET Provider	IsolationLevel in Connection Object Properties (For more information about ADO .NET, see System i Access.)

These isolation levels are supported by automatically locking the appropriate data. Depending on the type of lock, this limits or prevents access to the data by concurrent activation groups that use different commitment definitions. Each database manager supports at least two types of locks:

Share Limits concurrent activation groups that use different commitment definitions to read-only operations on the data.

Exclusive
Prevents concurrent activation groups using different commitment definitions from updating or deleting the data. Prevents concurrent activation groups using different commitment definitions that are running COMMIT(*RS), COMMIT(*CS), or COMMIT(*RR) from reading the data. Concurrent activation groups using different commitment definitions that are running COMMIT(*UR) or COMMIT(*NC) are allowed to read the data.

The following descriptions of isolation levels refer to locking data in row units. Individual implementations can lock data in larger physical units than base table rows. However, logically, locking occurs at the base-table row level across all products. Similarly, a database manager can escalate a lock to a higher level. An activation group is guaranteed at least the minimum requested lock level.

For a detailed description of record lock durations, see the discussion and table in the Commitment control topic of the SQL Programming topic collection.

DB2 for i supports five isolation levels. For all isolation levels except No Commit, the database manager places exclusive locks on every row that is inserted, updated, or deleted. This ensures that any row changed during a unit of work is not changed by any other activation group that uses a different commitment definition until the unit of work is complete.

Repeatable read

The Repeatable Read (RR) isolation level ensures:

- Any row read during a unit of work is not changed by other activation groups that use different commitment definitions until the unit of work is complete.

5. For **WITH HOLD** cursors, these rules apply to when the rows were actually read. For read-only **WITH HOLD** cursors, the rows may have actually been read in a prior unit of work.

- Any row changed (or a row that is currently locked with an UPDATE row lock) by another activation group using a different commitment definition cannot be read until it is committed.

In addition to any exclusive locks, an activation group running at level RR acquires at least share locks on all the rows it reads. Furthermore, the locking is performed so that the activation group is completely isolated from the effects of concurrent activation groups that use different commitment definitions.

In the SQL 2003 Core standard, Repeatable Read is called Serializable.

DB2 for i supports repeatable read through COMMIT(*RR). Repeatable read isolation level is supported by locking the tables containing any rows that are read or updated.

Read stability

Like level RR, level Read Stability (RS) ensures that:

- Any row read during a unit of work is not changed by other activation groups that use different commitment definitions until the unit of work is complete.
- Any row changed (or a row that is currently locked with an UPDATE row lock) by another activation group using a different commitment definition cannot be read until it is committed.

Unlike RR, RS does not completely isolate the activation group from the effects of concurrent activation groups that use a different commitment definition. At level RS, activation groups that issue the same query more than once might see additional rows. These additional rows are called *phantom rows*.

For example, a phantom row can occur in the following situation:

1. Activation group P1 reads the set of rows *n* that satisfy some search condition.
2. Activation group P2 then INSERTs one or more rows that satisfy the search condition and COMMITs those INSERTs.
3. P1 reads the set of rows again with the same search condition and obtains both the original rows and the rows inserted by P2.

In addition to any exclusive locks, an activation group running at level RS acquires at least share locks on all the rows it reads.

In the SQL 2003 Core standard, Read Stability is called Repeatable Read.

DB2 for i supports read stability through COMMIT(*ALL) or COMMIT(*RS).

Cursor stability

Like levels RR and RS, level Cursor Stability (CS) ensures that any row that was changed (or a row that is currently locked with an UPDATE row lock) by another activation group using a different commitment definition cannot be read until it is committed. Unlike RR and RS, level CS only ensures that the current row of every updatable cursor is not changed by other activation groups using different commitment definitions. Thus, the rows that were read during a unit of work can be changed by other activation groups that use a different commitment definition.

---

6. For **WITH HOLD** cursors, these rules apply to when the rows were actually read. For read-only **WITH HOLD** cursors, the rows may have actually been read in a prior unit of work.

In addition to any exclusive locks, an activation group running at level CS may acquire a share lock for the current row of every cursor.

In the SQL 2003 Core standard, Cursor Stability is called Read Committed.

DB2 for i supports cursor stability through COMMIT(*CS).

Uncommitted read

For a SELECT INTO, a FETCH with a read-only cursor, subquery, or fullselect used in an INSERT statement, level Uncommitted Read (UR) allows:

- Any row read during the unit of work to be changed by other activation groups that run under a different commitment definition.
- Any row changed (or a row that is currently locked with an UPDATE row lock) by another activation group running under a different commitment definition to be read even if the change has not been committed.

For other operations, the rules of level CS apply.

In the SQL 2003 Core standard, Uncommitted Read is called Read Uncommitted.

DB2 for i supports uncommitted read through COMMIT(*CHG) or COMMIT(*UR).

No commit

For all operations, the rules of level UR apply to No Commit (NC) except:

- Commit and rollback operations have no effect on SQL statements. Cursors are not closed, and LOCK TABLE locks are not released. However, connections in the release-pending state are ended.
- Any changes are effectively committed at the end of each successful change operation and can be immediately accessed or changed by other application groups using different commitment definitions.

DB2 for i supports No Commit through COMMIT(*NONE) or COMMIT(*NC).

Note: *(For distributed applications.)* When a requested isolation level is not supported by an application server, the isolation level is escalated to the next highest supported isolation level. For example, if RS is not supported by an application server, the RR isolation level is used.

Comparison of isolation levels

The following table summarizes information about isolation levels.

	NC	UR	CS	RS	RR
Can the application see uncommitted changes made by other application processes?	Yes	Yes	No	No	No
Can the application update uncommitted changes made by other application processes?	No	No	No	No	No
Can the re-execution of a statement be affected by other application processes? *See phenomenon P3 (phantom) below.*	Yes	Yes	Yes	Yes	No
Can “updated” rows be updated by other application processes?	Yes	No	No	No	No

	NC	UR	CS	RS	RR
Can "updated" rows be read by other application processes that are running at an isolation level other than UR and NC?	Yes	No	No	No	No
Can "updated" rows be read by other application processes that are running at the UR or NC isolation level?	Yes	Yes	Yes	Yes	Yes
Can "accessed" rows be updated by other application processes? For RS, "accessed rows" typically means rows selected. For RR, see the product-specific documentation. *See phenomenon P2 (nonrepeatable read) below.*	Yes	Yes	Yes	No	No
Can "accessed" rows be read by other application processes?	Yes	Yes	Yes	Yes	Yes
Can "current" row be updated or deleted by other application processes? *See phenomenon P1 (dirty-read) below.*	See Note below	See Note below	See Note below	No	No

Note: This depends on whether the cursor that is positioned on the "current" row is updatable:

- If the cursor is updatable, the current row cannot be updated or deleted by other application processes
- If the cursor is not updatable,
  - For UR or NC, the current row can be updated or deleted by other application processes.
  - For CS, the current row may be updatable in some circumstances.

Examples of Phenomena:

P1 *Dirty Read.* Unit of work UW1 modifies a row. Unit of work UW2 reads that row before UW1 performs a COMMIT. UW1 then performs a ROLLBACK. UW2 has read a nonexistent row.

P2 *Nonrepeatable Read.* Unit of work UW1 reads a row. Unit of work UW2 modifies that row and performs a COMMIT. UW1 then re-reads the row and obtains the modified data value.

P3 *Phantom.* Unit of work UW1 reads the set of *n* rows that satisfies some search condition. Unit of work UW2 then INSERTs one or more rows that satisfies the search condition. UW1 then repeats the initial read with the same search condition and obtains the original rows plus the inserted rows.

Storage Structures

The IBM i product is an object-based system. All database objects in DB2 for i (tables and indexes for example) are objects in the IBM i operating system. The single-level storage manager manages all storage of objects of the database, so database specific storage structures (for example, table spaces) are unnecessary.

A distributed table allows data to be spread across different database partitions. The partitions included are determined by the nodegroup specified when the table is created or altered. A nodegroup is a group of one or more IBM i products. A partitioning map is associated with each nodegroup. The partitioning map is used by the database manager to determine which system from the nodegroup will store a given row of data. For more information about nodegroups and data partitioning see the DB2 Multisystem book.

A table can also include columns that register links to data that are stored in external files. The mechanism for this is the DataLink data type. A DataLink value which is recorded in a regular table points to a file that is stored in an external file server.

The DB2 File Manager on a file server works in conjunction with DB2 to provide the following optional functionality:

- Referential integrity to ensure that files currently linked to DB2 are not deleted or renamed.
- Security to ensure that only those with suitable SQL privileges on the DataLink column can read the files linked to that column.

The DataLinker comprises the following facilities:

DataLinks File Manager
Registers all the files in a particular file server that are linked to DB2.

DataLinks Filter
Filters file system commands to ensure that registered files are not deleted or renamed. Optionally, filters commands to ensure that proper access authority exists.

Character conversion

A *string* is a sequence of bytes that may represent characters. Within a string, all the characters are represented by a common coding representation. In some cases, it might be necessary to convert these characters to a different coding representation. The process of conversion is known as *character conversion*.

Character conversion can occur when an SQL statement is executed remotely.[7] Consider, for example, these two cases:

- The values of variables sent from the application requester to the current server.
- The values of result columns sent from the current server to the application requester.

In either case, the string could have a different representation at the sending and receiving systems. Conversion can also occur during string operations on the same system.

Note that SQL statements are strings and are therefore subject to character conversion.

The following list defines some of the terms used when discussing character conversion.

character set
A defined set of characters. For example, the following character set appears in several code pages:
- 26 non-accented letters A through Z
- 26 non-accented letters a through z
- digits 0 through 9

7. Character conversion, when required, is automatic and is transparent to the application when it is successful. A knowledge of conversion is, therefore, unnecessary when all the strings involved in a statement's execution are represented in the same way. Thus, for many readers, character conversion may be irrelevant.

- . , : ; ? ( ) ' " / - _ & + = < >

code page
A set of assignments of characters to code points. In EBCDIC, for example, "A" is assigned code point X'C1' and "B" is assigned code point X'C2'. Within a code page, each code point has only one specific meaning.

code point
A unique bit pattern that represents a character within a code page.

coded character set
A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.

encoding scheme
A set of rules used to represent character data. For example:

- Single-byte EBCDIC
- Single-byte ASCII
- Double-byte EBCDIC
- Mixed single- and double-byte ASCII[8]
- Unicode (UTF-8, UCS-2, and UTF-16 universal coded character sets).

substitution character
A unique character that is substituted during character conversion for any characters in the source coding representation that do not have a match in the target coding representation.

Unicode
A universal encoding scheme for written characters and text that enables the exchange of data internationally. It provides a character set standard that can be used all over the world. It uses a 16-bit encoding form that provides code points for more than 65,000 characters and an extension called UTF-16 that allows for encoding as many as a million more characters. It provides the ability to encode all characters used for the written languages of the world and treats alphabetic characters, ideographic characters, and symbols equivalently because it specifies a numeric value and a name for each of its characters. It includes punctuation marks, mathematical symbols, technical symbols, geometric shapes, and dingbats. Three encoding forms are supported:

- UTF-8: Unicode Transformation Format, a 8-bit encoding form designed for ease of use with existing ASCII-based systems. UTF-8 data is stored in character data types. The CCSID value for data in UTF-8 format is 1208.

  A UTF-8 character can be 1,2,3 or 4 bytes in length. A UTF-8 data string can contain any combination of SBCS and DBCS data, including surrogates and combining characters.
- UCS-2: Universal Character Set coded in 2 octets, which means that characters are represented in 16-bits per character. UCS-2 data is stored in graphic data types. The CCSID value for data in UCS-2 format is 13488.

8. UTF-8 unicode data is also mixed data. In this book, however, mixed data refer to mixed single- and double-byte data.

UCS-2 is a subset of UTF-16. UCS-2 is identical to UTF-16 except that UTF-16 also supports combining characters and surrogates. Since UCS-2 is a simpler form of UTF-16, UCS-2 data will typically perform better than UTF-16.[9]

- UTF-16: Unicode Transformation Format, a 16-bit encoding form designed to provide code values for over a million characters. UTF-16 data is stored in graphic data types. The CCSID value for data in UTF-16 format is 1200.

  Both UTF-8 and UTF-16 data can contain *combining characters*. Combining character support allows a resulting character to be comprised of more than one character. After the first character, hundreds of different non-spacing accent characters (umlauts, accents, etc.) can follow in the data string. The resulting character may already be defined in the character set. In this case, there are multiple representations for the same character. For example, in UTF-16, an *é* can be represented either by X'00E9' (the normalized representation) or X'00650301' (the non-normalized combining character representation).

  Since multiple representations of the same character will not compare equal, it is usually not a good idea to store both forms of the characters in the database. *Normalization* is a process that replaces the string of combining characters with equivalent characters that do not include combining characters. After normalization has occurred, only one representation of any specific character will exist in the data. For example, in UTF-16, any instances of X'00650301' (the non-normalized combining character representation of *é*) will be converted to X'00E9' (the normalized representation of *é*).[10]

  In order to properly handle UTF-8 in predicates, normalization may occur.

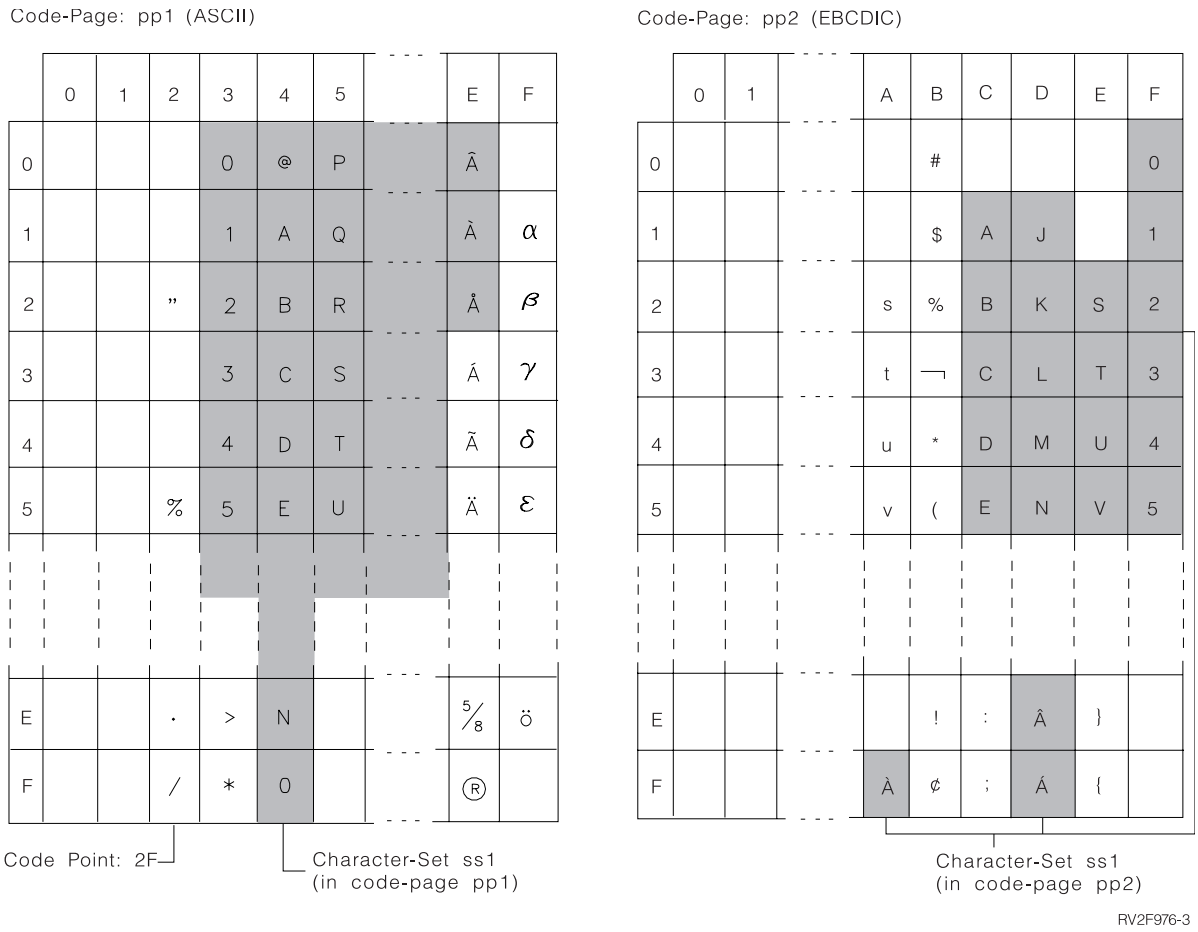
  Both UTF-8 and UTF-16 can contain 4 byte characters called *surrogates*. Surrogates are 4 byte sequences that can address one million more characters than would be available in a 2 byte character set.

Character sets and code pages

The following example shows how a typical character set might map to different code points in two different code pages.

9. UCS-2 can contain surrogates and combining characters, however, they are not recognized as such. Each 16–bits is considered to be a character.

10. Since normalization can significantly affect performance (from 2.5 to 25 percent extra CPU), the default in column definitions is NOT NORMALIZED.



Even with the same encoding scheme there are many different coded character sets, and the same code point can represent a different character in different coded character sets. Furthermore, a byte in a character string does not necessarily represent a character from a single-byte character set (SBCS). Character strings are also used for mixed data (a mixture of single-byte characters and double-byte characters) and for data that is not associated with any character set (called bit data). This is not the case with graphic strings; the database manager assumes that every pair of bytes in every graphic string represents a character from a double-byte character set (DBCS) or universal coded character set (UCS-2 or UTF-16).

A coded character set identifier (CCSID) in a native encoding scheme is one of the coded character sets in which data may be stored at that site. A CCSID in a foreign encoding scheme is one of the coded character sets in which data cannot be stored at that site. For example, DB2 for i can store data in a CCSID with an EBCDIC encoding scheme, but not in an ASCII encoding scheme.

A variable containing data in a foreign encoding scheme (other than Unicode) is always converted to a CCSID in the native encoding scheme when the variable is used in a function or in the *select-list*. A variable containing data in a foreign encoding scheme is also effectively converted to a CCSID in the native encoding scheme when used in comparison or in an operation that combines strings. Which CCSID in the native encoding scheme the data is converted to is based on the foreign CCSID and the default CCSID.

For details on character conversion, see:

- "Conversion rules for assignments" on page 104
- "Conversion rules for comparison:" on page 112
- "Conversion rules for operations that combine strings" on page 120
- "Data representation considerations" on page 46.

Coded character sets and CCSIDs

IBM's Character Data Representation Architecture (CDRA) deals with the differences in string representation and encoding. The *Coded Character Set Identifier (CCSID)* is a key element of this architecture. A CCSID is a 2-byte (unsigned) binary number that uniquely identifies an encoding scheme and one or more pairs of character sets and code pages.

A CCSID is an attribute of strings, just as length is an attribute of strings. All values of the same string column have the same CCSID.

Character conversion is described in terms of CCSIDs of the source and target. Each database manager provides support to identify valid source and target combinations and to perform the conversion from one coded character set to another. In some cases, no conversion is necessary even though the strings involved have different CCSIDs.

Different types of conversions may be supported by the database manager. Round-trip conversions attempt to preserve characters in one CCSID that are not defined in the target CCSID so that if the data is subsequently converted back to the original CCSID, the same original characters result. Enforced subset match conversions do not attempt to preserve such characters. For more information, see IBM's Character Data Representation Architecture (CDRA).

Default CCSID

Every application server and application requester has a default CCSID (or default CCSIDs in installations that support DBCS data).

The CCSID of the following types of strings is determined at the current server:

- String constants (including string constants that represent datetime values) when the CCSID of the source is in a foreign encoding scheme
- Special registers with string values (such as USER and CURRENT SERVER)
- CAST specifications where the result is a character or graphic string
- Results of CHAR, DATAPARTITIONNAME, DAYNAME, DBPARTITIONNAME, DIGITS, HEX, MONTHNAME, SOUNDEX, SPACE, and VARCHAR_FORMAT scalar functions
- Results of DECRYPT_CHAR, DECRYPT_DB, CHAR, GRAPHIC, VARCHAR, and VARGRAPHIC scalar functions when a CCSID is not specified as an argument
- Results of the CLOB and DBCLOB scalar functions when a CCSID is not specified as an argument[11]

11. If the default CCSID is 65535, the CCSID used will be the value of the DFTCCSID job attribute (or an associated CCSID of the DFTCCSID). If there is no associated mixed data CCSID,
    - the CCSID used when FOR MIXED DATA is specified will be 65535,
    - the CCSID for GRAPHIC and VARGRAPHIC will be 65535, and
    - the CCSID for DBCLOB will be 1200.

- String columns defined by the CREATE TABLE or ALTER TABLE statements when an explicit CCSID is not specified for the column[11]
- String columns defined by the DECLARE GLOBAL TEMPORARY TABLE statement when an explicit CCSID is not specified for the column[11]
- Distinct types when the source type is a character or graphic string type
- String parameters defined by CREATE FUNCTION or CREATE PROCEDURE statements when an explicit CCSID is not specified for the parameter [11]

If one of the types of strings above is used in a CREATE VIEW statement, the default CCSID is determined at the time the view is created.

In a distributed application, the default CCSID of variables is determined by the application requester. In a non-distributed application, the default CCSID of variables is determined by the application server. On the IBM i operating system, the default CCSID is determined by the CCSID job attribute. For more information about CCSIDs, see the Work with CCSIDs topic in the Globalization topic collection.

Collating sequence

A collating sequence (also called a sort sequence) defines how characters in a character set relate to each other when they are compared and ordered.

Different collating sequences are useful for those who want their data ordered for a specific language. For example, lists can be ordered as they are normally seen for a specific language. A collating sequence can also be used to treat certain characters as equivalent, for instance, **a** and **A**. A collating sequence works on all comparisons that involve:

- SBCS character data (including bit data)
- the SBCS portion of mixed data
- Unicode data (UTF-8, UCS-2, or UTF-16).

SBCS collating sequence support is implemented using a 256-byte table. Each byte in the table corresponds to a code point or character in a SBCS code page. Because the collating sequence is applicable to character data, a CCSID must be associated with the table. The bytes in the collating sequence table are set based on how each code point is to compare to other code points in that code page. For example, if the characters **a** and **A** are to be treated as equivalents for comparisons, the bytes in the collating sequence table for their code points contain the same value, or weight.

UCS-2 collating sequence support is implemented using a multi-byte table. A pair of bytes within the table corresponds to a character in the UCS-2 code page. Only a subset of the thousands of characters in UCS-2 are typically represented in the table. Only those characters that are to compare differently (and possibly other characters in the same ward) will be represented in the table. The bytes in the collating sequence table are set based on how each character is to compare with other characters in UCS-2.

When two or more bytes (or pair of bytes for UCS-2) in a collating sequence table have the same value, the collating sequence is a shared-weight collating sequence. If every byte (or pair of bytes for UCS-2) in a collating sequence table has a unique value, the collating sequence is a unique-weight collating sequence. For many languages, unique- and shared-weight collating sequences are shipped on the

system as part of the operating system. If you need collating sequences for other languages or needs, you define them using the Create Table (CRTTBL) command.

UTF-8 and UTF-16 collating sequence support is implemented using ICU (International Components for Unicode). This is a standard API to sort Unicode. The API produces the same result for normalized and non-normalized data and returns a sort weight based on language specific rules. The IBM i operating system supports ICU 2.3.1, ICU 3.4, and ICU 4.0 collating sequences, but ICU 3.4 or ICU 4.0 should be used. The ICU collating sequence table I34en_us (United States locale) will sort data differently than I34fr_FR (French locale).

If ICU is used, the LIKE predicate and the LOCATE, POSITION, POSSTR, and POSITION scalar functions are not supported.

If ICU 2.3.1 is used, the query cannot contain:

- EXCEPT or INTERSECT operations,
- VALUES in a fullselect,
- OLAP specifications,
- recursive common table expressions,
- ORDER OF,
- scalar fullselects (scalar subselects are supported),
- full outer join,
- LOBs in a GROUP BY,
- grouping sets or super groups,
- ORDER BY or FETCH FIRST n ROWS clause in a subselect,
- CONTAINS or SCORE functions,
- XMLAGG, XMLATTRIBUTES, XMLCOMMENT, XMLCONCAT, XMLDOCUMENT, XMLELEMENT, XMLFOREST, XMLGROUP, XMLNAMESPACES, XMLPI, XMLROW, or XMLTEXT functions,
- global variables, or
- references to arrays.

An ICU collating sequence table will generally produce results that are more culturally correct, however:

- The performance of SQL statements that use an ICU collating sequence table will generally perform worse than when using either an SBCS or UCS-2 collating sequence table. Indexes can be created with an ICU collating sequence table, however, to improve performance. In this case, the index key values will contain the ICU weighted value which will greatly reduce the number of times the system's ICU support is called.
- The storage necessary for indexes that use an ICU collating sequence table will generally be greater than when using either an SBCS or UCS-2 collating sequence table. The key values can be up to 3 times longer than the length of SBCS data used to produce the key and up to 6 times longer than the length of DBCS data used to produce the key.

It is important to remember that the data itself is not altered by the collating sequence. Instead, a weighted representation of the data is used for the comparison. In SQL, a collating sequence is specified on the CRTSQLxxx, STRSQL, and RUNSQLSTM commands. The SET OPTION statement can be used to specify the collating sequence within the source of a program containing embedded SQL. The collating sequence applies to all character comparisons performed in the SQL

statements. The default collating sequence on the system is the internal sequence that occurs when the hexadecimal representation of characters are used. This is the sequence you get when the SRTSEQ(*HEX) is specified. For programs precompiled with a release of the product that is earlier than Version 2 Release 3, the collating sequence is *HEX.

Collating sequences do not apply to FOR BIT DATA or binary string columns.

The collating sequence is not allowed for an index or unique constraint that contains a key column with a field procedure.

The *collating sequence* is explicitly specified through the following interfaces:

Table 3. Collating Sequence Interfaces

SQL Interface	Specification
Embedded SQL	SRTSEQ parameter on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to set the SRTSEQ values. (For more information about CRTSQLxxx commands, see Embedded SQL Programming.)
Run SQL Statements	SRTSEQ parameter on the Run SQL Statements (RUNSQLSTM) command. (For more information about the RUNSQLSTM command, see SQL Programming.)
Call Level Interface (CLI) on the server	SQL_ATTR_JOB_SORT_SEQUENCE environment variable (For more information about CLI, see SQL Call Level Interfaces (ODBC).)
JDBC or SQLJ on the server using IBM Developer Kit for Java	job.sort.sequence property object (For more information about JDBC and SQLJ, see IBM Developer Kit for Java.)
ODBC on a client using the IBM i Access Family ODBC Driver	Sort Type in ODBC Setup (For more information about ODBC, see System i Access.)
JDBC on a client using the IBM Toolbox for Java	Sort Sequence Table in JDBC Setup (For more information about JDBC, see System i Access.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java.)
OLE DB on a client using the IBM i Access Family OLE DB Provider	Sort Sequence Connection Object Properties (For more information about OLE DB, see System i Access.)
ADO .NET on a client using the IBM i Access Family ADO .NET Provider	SortSequence in Connection Object Properties (For more information about ADO .NET, see System i Access.)

For more information about CCSIDs, see the Work with CCSIDs topic in the Globalization section of the IBM i Information Center. For more information about collating sequences and the sequences shipped with the system, see the DB2 and SQL collating sequence topic in the IBM i Information Center.

Distributed relational database

A *distributed relational database* consists of a set of tables and other objects that are spread across different but interconnected computer systems or logical partitions on the same computer system. Each computer system has a relational database manager that manages the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a database manager to execute SQL statements on another computer system.

Distributed relational databases are built on formal requester-server protocols and functions. An *application requester* supports the application end of a connection. It transforms a database request from the application into communication protocols suitable for use in the distributed database network. These requests are received and processed by an *application server* at the other end of the connection.[12] Working together, the application requester and application server handle the communication and location considerations so that the application is isolated from these considerations and can operate as if it were accessing a local database. A simple distributed relational database environment is illustrated in Figure 8.

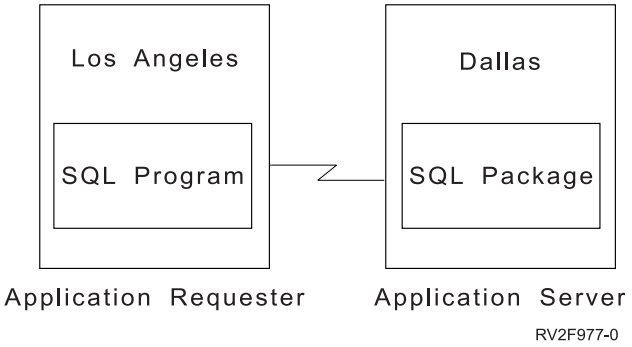


Figure 8. A Distributed Relational Database Environment

For more information about Distributed Relational Database Architecture™ (DRDA) communication protocols, see Open Group Publications: DRDA Vol. 1: Distributed Relational Database Architecture (DRDA)

Application servers

An activation group must be connected to the application server of a database manager before SQL statements can be executed.

A *connection* is an association between an activation group and a local or remote application server. A connection is also known as a session or an SQL session. Connections are managed by the application. The CONNECT statement can be used to establish a connection to an application server and make that application server the current server of the activation group.

An implicit CONNECT operation may also establish a connection to the application server:

12. This is also known as a *an application server*.

- An implicit CONNECT operation may occur when invoking a program, service program, or the STRSQL command. The application server is determined by the RDB parameter on the CRTSQLxxx and STRSQL commands.

  In the these cases, the implicit CONNECT operation will not occur if an implicit or explicit CONNECT operation has already successfully or unsuccessfully occurred in the activation group. Thus, an activation group cannot be implicitly connected to an application server more than once.
- An implicit CONNECT operation may occur when using a three-part object name or an alias that is defined to reference a three-part name of a table or view.

  In the these cases, an implicit CONNECT operation is only allowed if all the objects referenced in the SQL statement refer to the same relational database. The only exceptions are:
  - The target table of an INSERT statement may be in one relational database and the tables referenced in the *select-statement* of the INSERT may be in another relational database.
  - The new table for a CREATE TABLE or DECLARE GLOBAL TEMPORARY TABLE statement may be in one relational database and the tables referenced in the *select-statement* may be in another relational database.

  The implicit CONNECT changes the current server for the statement. At the end of the statement the connection is set back to the prior current server.
- When creating three-part qualified SQL stored procedures or SQL functions, objects in the procedure body must exist on both the DRDA client and DRDA server. If the procedure body objects do not exist, object not found errors may be signalled.

An application server can be local to, or remote from, the environment where the activation group is started. (An application server is present, even when distributed relational databases are not used.) This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement. For more information about the directory, see the relational database folders in System i Navigator or the directory commands (ADDRDBDIRE, CHGRDBDIRE, DSPRDBDIRE, RMVRDBDIRE, and WRKRDBDIRE) in the following IBM i Information Center topics:

- SQL Programming
- Distributed Database Programming
- CL commands

To execute a static SQL statement that references tables or views, an application server uses the bound form of the statement. This bound statement is taken from a package that the database manager previously created through a bind operation. The appropriate package is determined by the combination of:

- The name of the package specified by the SQLPKG parameter on the CRTSQLxxx commands. See Embedded SQL Programming for a description of the CRTSQLxxx commands.
- The internal consistency token that makes certain the package and program were created from the same source at the same time.

A DB2 relational database product may support a feature that is not supported by the version of the DB2 product that is connecting to the application server. Some of these features are product-specific, and some are shared by more than one product.

For the most part, an application can use the statements and clauses that are supported by the database manager of the application server to which it is

currently connected, even though that application is running via the application requester of a database manager that does not support some of those statements and clauses. Restrictions are listed in Appendix B, "Characteristics of SQL statements," on page 1501.

CONNECT (Type 1) and CONNECT (Type 2)

There are two types of CONNECT statements with the same syntax but different semantics. CONNECT (Type 1) is used for remote unit of work. CONNECT (Type 2) is used for distributed unit of work.

See "CONNECT (Type 1) and CONNECT (Type 2) differences" on page 1511 for a summary of the differences.

Remote unit of work

The *remote unit of work* facility provides for the remote preparation and execution of SQL statements. An activation group at computer system A can connect to an application server at computer system B. Then, within one or more units of work, that activation group can execute any number of static or dynamic SQL statements that reference objects at B. After ending a unit of work at B, the activation group can connect to an application server at computer system C, and so on.

Most SQL statements can be remotely prepared and executed with the following restrictions:

- All objects referenced in a single SQL statement must be managed by the same application server.
- All of the SQL statements in a unit of work must be executed by the same application server.

Remote unit of work connection management

An activation group is in one of three states at any time:

- Connectable and connected
- Unconnectable and connected
- Connectable and unconnected

The following diagram shows the state transitions:

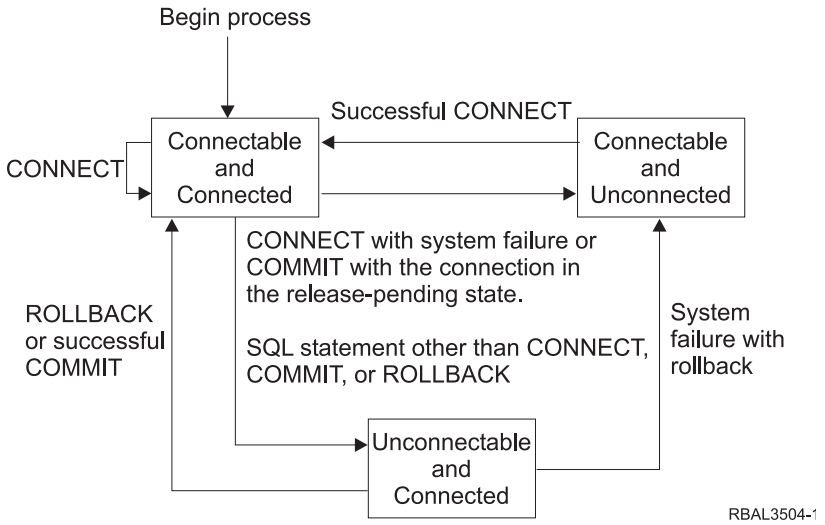


Figure 9. Remote Unit of Work Activation Group Connection State Transition

The initial state of an activation group is *connectable* and *connected*.

The connectable and connected state

An activation group is connected to an application server and CONNECT statements can be executed. The activation group enters this state when it completes a rollback or successful commit from the unconnectable and connected state, or a CONNECT statement is successfully executed from the connectable and unconnected state.

The unconnectable and connected state

An activation group is connected to an application server, but a CONNECT statement cannot be successfully executed to change application servers. The activation group enters this state from the connectable and connected state when it executes any SQL statement other than CONNECT, COMMIT, or ROLLBACK.

The connectable and unconnected state

An activation group is not connected to an application server. The only SQL statement that can be executed is CONNECT.

The activation group enters this state when:

- The connection was previously released and a successful COMMIT is executed.
- The connection is disconnected using the SQL DISCONNECT statement.
- The connection was in a connectable state, but the CONNECT statement was unsuccessful.

Consecutive CONNECT statements can be executed successfully because CONNECT does not remove the activation group from the connectable state. A CONNECT to the application server to which the activation group is currently connected is executed like any other CONNECT statement. CONNECT cannot execute successfully when it is preceded by any SQL statement other than CONNECT, COMMIT, DISCONNECT, SET CONNECTION, RELEASE, or ROLLBACK (unless running with COMMIT(*NC)). To avoid an error, execute a commit or rollback operation before a CONNECT statement is executed.

Application-directed distributed unit of work

The *application-directed distributed unit of work facility* also provides for the remote preparation and execution of SQL statements in the same fashion as remote unit of work. Like remote unit of work, an activation group at computer system A can connect to an application server at computer system B and execute any number of static or dynamic SQL statements that reference objects at B before ending the unit of work. All objects referenced in a single SQL statement must be managed by the same application server. However, unlike remote unit of work, any number of application servers can participate in the same unit of work. A commit or rollback operation ends the unit of work.

Distributed unit of work is fully supported for APPC and TCP/IP connections.

Application-directed distributed unit of work connection management

At any time:

- An activation group is always in the *connected* or *unconnected* state and has a set of zero or more connections. Each connection of an activation group is uniquely identified by the name of the application server of the connection.
- An SQL connection is always in one of the following states:
  - Current and held
  - Current and release-pending
  - Dormant and held
  - Dormant and release-pending

Initial state of an activation group:

An activation group is initially in the connected state and has exactly one connection. The initial state of a connection is *current and held*.

The following diagram shows the state transitions:

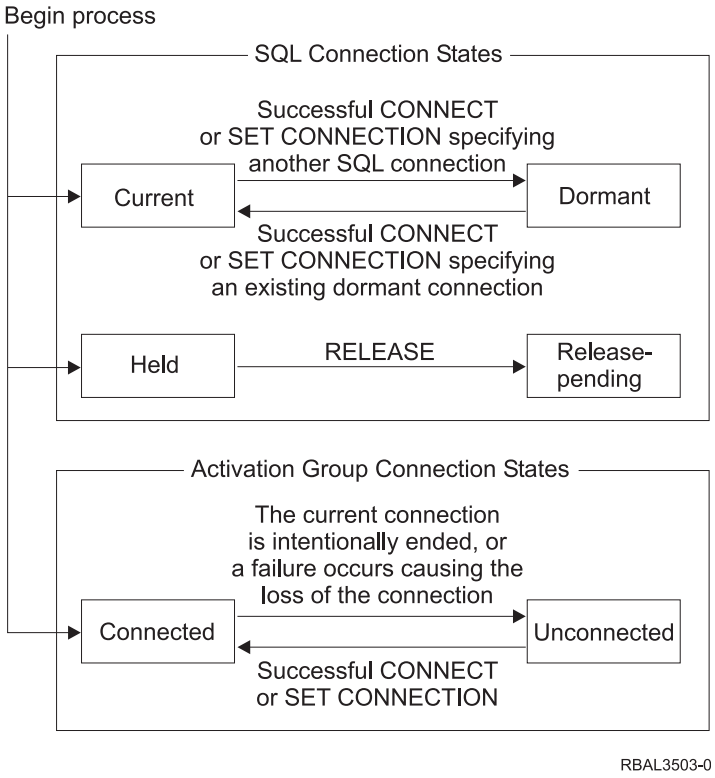


Figure 10. Application-Directed Distributed Unit of Work Connection and Activation Group Connection State Transitions

Connection states

If an application process successfully executes a CONNECT statement:

- The current connection is placed in the dormant state and held state.
- The server name is added to the set of connections and the new connection is placed in the current and held state.

If the server name is already in the set of existing connections of the activation group, an error is returned.

A connection in the dormant state is placed in the current state using the SET CONNECTION statement. When a connection is placed in the current state, the previous current connection, if any, is placed in the dormant state. No more than one connection in the set of existing connections of an activation group can be current at any time. Changing the state of a connection from current to dormant or from dormant to current has no effect on its held or release-pending state.

A connection is placed in the release-pending state by the RELEASE statement. When an activation group executes a commit operation, every release-pending connection of the activation group is ended. Changing the state of a connection from held to release-pending has no effect on its current or dormant state. Thus, a connection in the release-pending state can still be used until the next commit operation. There is no way to change the state of a connection from release-pending to held.

Activation group connection states

A different application server can be established by the explicit or implicit execution of a CONNECT statement. The following rules apply:

- An activation group cannot have more than one connection to the same application server at the same time.
- When an activation group executes a SET CONNECTION statement, the specified location name must be an existing connection in the set of connections of the activation group.
- When an activation group executes a CONNECT statement, the specified server name must not be an existing connection in the set of connections of the activation group.

If an activation group has a current connection, the activation group is in the *connected* state. The CURRENT SERVER special register contains the name of the application server of the current connection. The activation group can execute SQL statements that refer to objects managed by that application server.

An activation group in the unconnected state enters the connected state when it successfully executes a CONNECT or SET CONNECTION statement.

If an activation group does not have a current connection, the activation group is in the *unconnected* state. The CURRENT SERVER special register contents are equal to blanks. The only SQL statements that can be executed are CONNECT, DISCONNECT, SET CONNECTION, RELEASE, COMMIT, and ROLLBACK.

An activation group in the connected state enters the unconnected state when its current connection is intentionally ended or the execution of an SQL statement is unsuccessful because of a failure that causes a rollback operation at the current server and loss of the connection. Connections are intentionally ended when an activation group successfully executes a commit operation and the connection is in the release-pending state, or when an application process successfully executes the DISCONNECT statement.

When a connection is ended

When a connection is ended, all resources that were acquired by the activation group through the connection and all resources that were used to create and maintain the connection are deallocated. For example, if application process P has placed the connection to application server X in the application server state, all cursors of P at X will be closed and deallocated when the connection is ended during the next commit operation.

A connection can also be ended as a result of a communications failure in which case the activation group is placed in the unconnected state. All connections of an activation group are ended when the activation group ends.

Data representation considerations

Different systems represent data in different ways. When data is moved from one system to another, data conversion must sometimes be performed. Products supporting DRDA will automatically perform any necessary conversions at the receiving system.

With numeric data, the information needed to perform the conversion is the data type and the sending system's environment type. For example, when a

floating-point variable from a DB2 for i application requester is assigned to a column of a table at an z/OS® application server, the number is converted from IEEE format to System/370* format.

With character and graphic data, the data type and the environment type of the sending system are not sufficient. Additional information is needed to convert character and graphic strings. String conversion depends on both the coded character set of the data and the operation to be done with that data. String conversions are done in accordance with the IBM Character Data Representation Architecture (CDRA). For more information about character conversion, refer to the book *Character Data Representation Architecture Reference and Registry*, SC09-2190.

Distributed relational database

Chapter 2. Language elements

This section defines the basic syntax of SQL and language elements that are common to many SQL statements.

Characters

The basic symbols of keywords and operators in the SQL language are single-byte characters that are part of all character sets supported by the IBM relational database products.

Characters of the language are classified as letters, digits, or special characters.[13]

A *letter* is any of the 26 uppercase (A through Z) and 26 lowercase (a through z) letters of the English alphabet. [14]

A *digit* is any of the characters 0 through 9.

A *special character* is any of the characters listed below:

	space or blank	-	minus sign
"	quotation mark or double-quote or double quotation mark	.	period
%	percent	/	slash
&	ampersand	:	colon
'	apostrophe or single quote or single quotation mark	;	semicolon
(	left parenthesis	<	less than
)	right parenthesis	=	equals
*	asterisk	>	greater than
+	plus sign	?	question mark
,	comma	_	underline or underscore
|	vertical bar[16]	^	caret
!	exclamation mark[15]	[	left bracket
{	left brace	]	right bracket
}	right brace	¬	not [15]

13. Note that if the SQL statement is encoded as Unicode data, all characters of the statement except for string constants will be converted to single-byte characters prior to processing. Tokens representing string constants may be processed as UTF-16 graphic strings without conversion to single-byte.
14. Letters also include three code points reserved as alphabetic extenders for national languages (#, @, and $ in the United States). These three code points should be avoided because they represent different characters depending on the CCSID.
15. Using the not symbol (¬) and the exclamation point symbol (!) might inhibit code portability between IBM relational database products. Avoid using them because they are variant characters. Instead of ¬= or != use <>. Instead of ¬> or !> use <=. Instead of ¬< or !< use >=.
16. Using the vertical bar (|) character might inhibit code portability between IBM relational database products. Use the CONCAT operator instead of the concatenation operator (||).

Tokens

The basic syntactical units of the language are called *tokens*. A token consists of one or more characters, excluding blanks, control characters, and characters within a string constant or delimited identifier. (These terms are defined later.)

Tokens are classified as *ordinary* or *delimiter* tokens:

- An *ordinary token* is a numeric constant, an ordinary identifier, a host identifier, or a keyword.

  Examples

  ```
  1       .1       +2       SELECT       E       3
  ```

- A *delimiter token* is a string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark (?) is also a delimiter token when it serves as a parameter marker, as explained under "PREPARE" on page 1293.

  Examples

  ```
  ,     'Myst Island'       "fld1"       =       .
  ```

Spaces:

A *space* is a sequence of one or more blank characters.

Control Characters:

A *control character* is a special character that is used for string alignment. The following table contains the control characters that are handled by the database manager:

Table 4. Control Characters

Control Character	EBCDIC Hex Value	Unicode Graphic Hex Value
Tab	05	U+0009
Form Feed	0C	U+000C
Carriage Return	0D	U+000D
New Line	15	U+0085
Line Feed (New line)	25	U+000A
DBCS Space	—	U+3000

Tokens, other than string constants and certain delimited identifiers, must not include a control character or space. A control character or space can follow a token. A delimiter token, a control character, or a space *must* follow every ordinary token. If the syntax does not allow a delimiter token to follow an ordinary token, then a control character or a space must follow that ordinary token. The following examples illustrate the rule that is stated in this paragraph.

Here are some examples of combinations of the above ordinary tokens that, in effect, change the tokens:

```
1.1       .1+2       SELECTE       .1E       E3       SELECT1
```

This demonstrates why ordinary tokens must be followed by a delimiter token or a space.

Here are some examples of combinations of the above ordinary tokens and the above delimiter tokens that, in effect, change the tokens:

```
1.      .3
```

The period (.) is a delimiter token when it is used as a separator in the qualification of names. Here the dot is used in combination with an ordinary token of a numeric constant. Thus, the syntax does not allow an ordinary token to be followed by a delimiter token. Instead, the ordinary token must be followed by a space.

If the decimal point has been defined to be the comma, as described in "Decimal point" on page 127, the comma is interpreted as a decimal point in numeric constants. Here are some examples of these numeric constants:

```
1,2       ,1       1,       1,e1
```

If '1,2' and '1,e1' are meant to be two items, both the ordinary token (1) and the delimiter token (,) must be followed by a space, to prevent the comma from being interpreted as a decimal point. Although the comma is usually a delimiter token, the comma is part of the number when it is interpreted as a decimal point. Therefore, the syntax does not allow an ordinary token (1) to be followed by a delimiter token (,). Instead, an ordinary token must be followed by a space.

Comments:

Dynamic SQL statements can include SQL comments. Static SQL statements can include host language comments or SQL comments. Comments may be specified wherever a space may be specified, except within a delimiter token or between the keywords EXEC and SQL. In Java, SQL comments are not allowed within embedded Java expressions. There are two types of SQL comments:

simple comments
Simple comments are introduced by two consecutive hyphens (--). Simple comments cannot continue past the end of the line. For more information, see "SQL comments" on page 709.

bracketed comments
Bracketed comments are introduced by /* and end with */. A bracketed comment can continue past the end of the line. For more information, see "SQL comments" on page 709.

Uppercase and Lowercase:

Any token in an SQL statement may include lowercase letters, but a lowercase letter in an ordinary token is folded to uppercase, except for variables in the C and Java languages, which have case-sensitive identifiers. Delimiter tokens are never folded to uppercase. Thus, the statement:

```
select * from EMP where lastname = 'Smith';
```

is equivalent, after folding, to:

```
SELECT * FROM EMP WHERE LASTNAME = 'Smith';
```

Identifiers

An *identifier* is a token used to form a name. An identifier in an SQL statement is an SQL identifier, a system identifier, or a host identifier.

Note: $, @, #, and all other variant characters should not be used in identifiers because the code points used to represent them vary depending on the CCSID of the string in which they are contained. If they are used, unpredictable results may occur. For more information about variant characters, see the DB2 and SQL sort sequence topic.

SQL identifiers

There are two types of SQL identifiers: *ordinary identifiers* and *delimited identifiers*.

- An *ordinary identifier* is an uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. Note that ordinary identifiers are converted to uppercase. An ordinary identifier should not be a reserved word. See Appendix I, "Reserved schema names and reserved words," on page 1817 for a list of reserved words. If a reserved word is used as an identifier in SQL, it should be specified in uppercase and should be a delimited identifier or specified in a variable.
- A *delimited identifier* is a sequence of one or more characters enclosed within SQL escape characters. The sequence must consist of one or more characters. Leading blanks in the sequence are significant. Trailing blanks in the sequence are not significant. The length of a delimited identifier does not include the two SQL escape characters. Note that delimited identifiers are not converted to uppercase. The escape character is the quotation mark (") except in the following cases where the escape character is the apostrophe ('):
  - Interactive SQL when the SQL string delimiter is set to the quotation mark in COBOL syntax checking statement mode
  - Dynamic SQL in a COBOL program when the CRTSQLCBL or CRTSQLCBLI parameter OPTION(*QUOTESQL) specifies that the string delimiter is the quotation mark (")
  - COBOL application program when the CRTSQLCBL or CRTSQLCBLI parameter OPTION(*QUOTESQL) specifies that the string delimiter is the quotation mark (")

  The following characters are not allowed within delimited identifiers:
  - X'00' through X'3F' and X'FF'

System identifiers

A system identifier is used to form the name of system objects in the IBM i operating system. There are two types of system identifiers: ordinary identifiers and delimited identifiers.

- The rules for forming a system ordinary identifier are identical to the rules for forming an SQL ordinary identifier.
- The rules for forming a system delimited identifier are identical to those for forming SQL delimited identifiers, except:
  - The following special characters are not allowed in a delimited system identifier:
    - A blank (X'40')
    - An asterisk (X'5C')
    - An apostrophe (X'7D')
    - A question mark (X'6F')
    - A quotation mark (X'7F')
  - The bytes required for the escape characters are included in the length of the identifier unless the characters within the delimiters would form an ordinary identifier.

For example, “PRIVILEGES” is in uppercase and the characters within the delimiters form an ordinary identifier; therefore, it has a length of 10 bytes and is a valid system name for a column. Alternatively, “privileges” is in lowercase, has a length of 12 bytes, and is not a valid system name for a column because the bytes required for the delimiters must be included in the length of the identifier.

Examples

```
WKLYSAL     WKLY_SAL     "WKLY_SAL"     "UNION"     "wkly_sal"
```

See “Naming conventions” on page 54 for information on the maximum length of identifiers.

Host identifiers

A *host-identifier* is a name declared in the host program.

The rules for forming a host-identifier are the rules of the host language; except that DBCS characters cannot be used. For example, the rules for forming a host-identifier in a COBOL program are the same as the rules for forming a user-defined word in COBOL. Names beginning with the characters 'SQ', 'SQL', 'sql', 'RDI', or 'DSN' should not be used because precompilers generate host variables that begin with these characters. In Java, do not use names beginning with `'__sJT_'`.

See Table 5 on page 62 for the limits on the maximum size of the host identifier name imposed by DB2 for i.

17. 'SQ' is allowed in C, COBOL, and PL/I; it should not be used in RPG.

Naming conventions

The rules for forming a name depend on the type of the object designated by the name and the naming option (*SQL or *SYS). The naming option is specified on the CRTSQLxxx, RUNSQLSTM, and STRSQL commands. The SET OPTION statement can be used to specify the naming option within the source of a program containing embedded SQL. The syntax diagrams use different terms for different types of names.

The following list defines these terms.

alias-name
A qualified or unqualified name that designates an alias. The qualified form of an *alias-name* depends on the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of an *alias-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

An *alias-name* can specify either the name of the alias or the system object name of the alias.

array-type-name
A qualified or unqualified name that designates an array type. The qualified form of a *array-type-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of a *array-type-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

For system naming, *array-type-names* cannot be qualified when used in a parameter data type of an SQL routine or in an SQL variable declaration in an SQL procedure.

authorization-name
A system identifier that designates a user or group of users. An *authorization-name* is a user profile name on the server. It must not be a delimited identifier that includes lowercase letters or special characters. See "Authorization IDs and authorization names" on page 68 for the distinction between an *authorization-name* and an authorization ID.

column-name
A qualified or unqualified name that designates a column of a table or a view. The unqualified form of a *column-name* is an SQL identifier. The qualified form is a qualifier followed by a period and an SQL identifier. The qualifier is a table name, a view name, or a correlation name.

For system naming, column names can be qualified using the form *schema-name/table-name.column-name* when the name is used in the COMMENT and LABEL statements. If column names need to be qualified

18. For system naming, the qualified form that uses a period is also accepted.

and correlation names are allowed in the statement, a correlation name can be used to qualify the column. The period form of qualification can also be used.

A *column-name* can specify either the column name or the system column name of a column of a table or view. If a *column-name* is delimited, the delimiters are considered to be part of the name when determining the length of the name.

constraint-name
A qualified or unqualified name that designates a constraint on a table. The qualified form of a *constraint-name* depends on the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of a *constraint-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

The implicit or explicit qualifier must be the same as the schema name of the table.

correlation-name
An SQL identifier that designates a table, a view, or individual rows of a table or view.

cursor-name
An SQL identifier that designates an SQL cursor.

descriptor-name
A variable name or string constant that designates an SQL descriptor area (SQLDA). A variable that designates an SQL descriptor area must not have an indicator variable. The form *:host-variable:indicator-variable* is not allowed. See "References to host variables" on page 149 for a description of a variable.

distinct-type-name
A qualified or unqualified name that designates a distinct type. The qualified form of a *distinct-type-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of a *distinct-type-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

For system naming, *distinct-type-names* cannot be qualified when used in a parameter data type of an SQL routine or in an SQL variable declaration in an SQL function, SQL procedure, or trigger.

external-program-name
A qualified name, unqualified name, or a character string that designates an external program. The qualified form of an *external-program-name* depends on the naming option. For SQL naming, the qualified form is a *system-schema-name* followed by a period (.) and a system identifier. For system naming, the qualified form is a *system-schema-name* followed by a slash (/) followed by a system identifier[18].

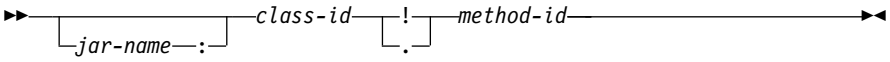
The unqualified form of an *external-program-name* is a system identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

For a service program name, the qualified form depends on the naming option. For SQL naming the qualified form is a *system-schema-name* followed by a period (.), followed by a system identifier for the service program name, followed by a left parenthesis, followed by an IBM i entry-point-name, followed by a right parenthesis (library-name.service-program-name(entry-point-name)). For system naming, the qualified form is a *system-schema-name* followed by a slash (/) followed by a system identifier for the service program name, followed by a left parenthesis, followed by an IBM i entry-point-name, followed by a right parenthesis (library-name/service-program-name(entry-point-name))[18]. If the entry point name contains lowercase characters, it must be enclosed in quotes.

The unqualified form of an service program name is a system identifier followed by a left parenthesis, followed by an IBM i entry-point-name, followed by a right parenthesis. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

The format of the character string form is either:

- An IBM i qualified program name ('library-name/program-name').
- An IBM i qualified source file name, followed by a left parenthesis, followed by an IBM i member name, and a right parenthesis ('library-name/source-file-name(member-name)'). This form is only valid when calling a REXX procedure.
- An IBM i qualified service program name, followed by a left parenthesis, followed by an IBM i entry-point-name, followed by a right parenthesis ('library-name/service-program-name(entry-point-name)').
- In Java, an optional *jar-name*, followed by a class identifier, followed by an exclamation point or period, followed by a method identifier ('class-id!method-id' or 'class-id.method-id').



jar-name

The *jar-name* is a case-sensitive string that identifies the jar schema when it was installed in the database. It can be either a simple identifier, or a schema qualified identifier. Examples are 'myJar' and 'myCollection.myJar'.

class-id

The *class-id* identifies the class identifier of the Java object. If the class is part of a Java package, the class identifier must include the complete Java package prefix. For example, if the class identifier is 'myPackage.StoredProcs', the Java virtual machine will look in the following directory for the StoredProcs class:

```
'/QIBM/UserData/OS400/SQLLib/
  Function/myPackage/StoredProcs/'
```

method-id

The *method-id* identifies the method name of the public, static Java method to be invoked.

This form is only valid for Java procedures and Java functions.

function-name
A qualified or unqualified name that designates a user-defined function, a cast function that was generated when a distinct type was created, or a built-in function. The qualified form of a *function-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

In a CREATE, COMMENT, DROP, GRANT, or REVOKE statement, the *schema-name* can be qualified with a *server-name*. In all other contexts, a *server-name* is not allowed.

The unqualified form of a *function-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

For system naming, functions names can only be qualified in the form *schema-name/function-name* when the name is used in a CREATE, COMMENT, DROP, GRANT, or REVOKE statement. The period form of qualification can be used in an expression.

host-label
A token that designates a label in a host program.

host-variable
A sequence of tokens that designates a host variable. A *host-variable* includes at least one *host-identifier*, as explained in "References to host variables" on page 149.

index-name
A qualified or unqualified name that designates an index. The qualified form of an *index-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of an *index-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

member-name
An unqualified identifier that designates a member of a database file. A member is also a partition of a partitioned table.

nodegroup-name
A qualified or unqualified name that designates a nodegroup. A nodegroup is a group of IBM i products across which a table will be distributed. For more information about distributed tables and nodegroups, see DB2 Multisystem.

The qualified form of a *nodegroup-name* depends on the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and a system identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by a system identifier[18].

The unqualified form of a *nodegroup-name* is a system identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

package-name
A qualified or unqualified name that designates a package. The qualified form of a *package-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and a system identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by a system identifier[18].

The unqualified form of a *package-name* is a system identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

parameter-name
An SQL identifier that designates a parameter for a function or procedure. If the *parameter-name* is for a procedure, the identifier may be preceded by a colon.

partition-name
An unqualified identifier that designates a partition of a partitioned table.

procedure-name
A qualified or unqualified name that designates a procedure. The qualified form of a *procedure-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of a *procedure-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

savepoint-name
An unqualified identifier that designates a savepoint.

schema-name
A qualified or unqualified name that provides a logical grouping for SQL objects. A schema name is used as a qualifier of the name of a table, view, index, procedure, function, trigger, sequence, variable, constraint, alias, type, or package. The unqualified form of a *schema-name* is a system identifier. The qualified form of a *schema-name* depends on the naming option.

For SQL naming, the unqualified schema name in an SQL statement is implicitly qualified by the *server-name*. The qualified form is a *server-name* followed by a (.) and a system identifier.

For system naming, the unqualified schema name in an SQL statement is implicitly qualified by the *server-name*. The qualified form is a *server-name* followed by a slash (/) and a system identifier[18].

If the *server-name* is used to qualify the name of the schema, the *server-name* may identify any supported remote server. Otherwise, the schema name is implicitly qualified with the current server.

Note: *schema-name* refers to either a schema created by the CREATE SCHEMA statement or to an IBM i library.

sequence-name
A qualified or unqualified name that designates a sequence. The qualified form of a *sequence-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name*

followed by a slash (/) followed by an SQL identifier[18]. For system naming, a *sequence-name* cannot be qualified with a slash when used in a NEXT VALUE or PREVIOUS VALUE expression (the slash-qualified form is only allowed in SQL schema statements). The period form of qualification can be used in a NEXT VALUE or PREVIOUS VALUE expression.

The unqualified form of a *sequence-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

A *sequence-name* can specify either the name of the sequence or the system object name of the sequence.

server-name

An SQL identifier that designates an application server. The identifier must start with a letter and must not include lowercase letters or special characters.

A *server-name* may be the actual name of the relational database or a relational database alias. For more information see the Add RDB Directory Entry (ADDRDBDIRE) command. However, if a three-part name is specified directly in an SQL statement (other than the base table specified in a CREATE ALIAS statement) it can use either the actual relational database name or the relational database alias name.

For example, if the actual name of the relational database is ABC and a relational database alias name of MYABC also references ABC.

```
SELECT * FROM ABC.SCHEMA1.T1   -- This is valid.

SELECT * FROM MYABC.SCHEMA1.T1 -- This is also valid.
```

specific-name

A qualified or unqualified name that uniquely identifies a procedure or function. The qualified form of a *specific-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of a *specific-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

SQL-condition-name

An SQL identifier that designates a condition in an SQL procedure, SQL function, or trigger body.

SQL-descriptor-name

A variable name or character or graphic string constant that designates an SQL descriptor that was allocated using the ALLOCATE DESCRIPTOR statement.

If a variable is used to designate the SQL descriptor:

- The variable must not be a CLOB or DBCLOB.
- If the variable is a graphic string, it must be a Unicode graphic string.
- The length of the contents of the variable must not exceed the maximum length for an *SQL-descriptor-name*.
- An indicator variable must not be specified. The form *:host-variable:indicator-variable* is not allowed.

- The contents of the variable are case-sensitive and are not converted to uppercase.

Leading and trailing blanks are trimmed from the variable or string. See "References to host variables" on page 149 for a description of a variable.

If a string constant is used to designate the SQL descriptor, the length of the constant must not exceed the maximum length for an *SQL-descriptor-name*.

SQL-label
An unqualified name that designates a label in an SQL procedure, SQL function, or trigger body. An *SQL-label* is an SQL identifier.

SQL-parameter-name
A qualified or unqualified name that designates a parameter in an SQL routine body. The unqualified form of an *SQL-parameter-name* is an SQL identifier. The qualified form is a *procedure-name* followed by a period (.) and an SQL identifier.

SQL-variable-name
A qualified or unqualified name that designates a variable in an SQL routine body. The unqualified form of an *SQL-variable-name* is an SQL identifier. The qualified form is an *SQL-label* followed by a period (.) and an SQL identifier.

statement-name
An SQL identifier that designates a prepared SQL statement.

system-column-name
An unqualified name that designates the IBM i column name of a table or a view. A *system-column-name* is a system identifier. *System-column-names* can be delimited identifiers, but the characters within the delimiters must not include lowercase letters or special characters.

system-object-name
An unqualified name that designates the IBM i name of a table, view, index, sequence, variable, or alias. A *system-object-name* is a system identifier.

If the unqualified name of the table, view, index, sequence, variable, or alias is a valid system identifier, the *system-object-name* of the table, view, index, sequence, variable, or alias is the unqualified name of the table, view, index, sequence, or alias.

system-schema-name
An unqualified name that designates the IBM i name of a schema. A *system-schema-name* is a system identifier.

If the unqualified name of the schema is a valid system identifier, the *system-schema-name* of the schema is the unqualified name of the schema.

table-name
A qualified or unqualified name that designates a table. The qualified form of a *table-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of a *table-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

A *table-name* can specify either the name of the table or the system object name of the table.

trigger-name
A qualified or unqualified name that designates a trigger on a table. The qualified form of a *trigger-name* depends on the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and a system identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of a *trigger-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

variable-name
A qualified or unqualified name that designates a global variable. The qualified form of a *variable-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18]. For system naming, a *variable-name* cannot be qualified with a slash when used in an expression (the slash-qualified form is only allowed in SQL schema statements). The period form of qualification can be used in an expression.

The unqualified form of a *variable-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

A *variable-name* can specify either the name of the variable or the system object name of the variable.

version-id
An identifier of 1 to 64 characters that is assigned to a package when the package is created. A *version-id* is only assigned when packages are created from a server other than DB2 for i.

view-name
A qualified or unqualified name that designates a view. The qualified form of a *view-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of a *view-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

A *view-name* can specify either the name of the view or the system object name of the view.

xsrobject-name
A qualified or unqualified name that designates an object in the XML schema repository. The qualified form of an *xsrobject-name* depends upon the naming option. For SQL naming, the qualified form is a *schema-name* followed by a period (.) and an SQL identifier. For system naming, the qualified form is a *schema-name* followed by a slash (/) followed by an SQL identifier[18].

The unqualified form of an *xsrobject-name* is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

Table 5. Identifier Length Limits (in bytes)

Identifier Type	Maximum Length
Longest authorization name[19]	10
Longest correlation name	128
Longest cursor name	128
Longest external program name (string form)	279
Longest external program name (unqualified form)[20]	10
Longest host identifier	64
Longest partition name	10
Longest savepoint name	128
Longest schema name	128
Longest server name	18
Longest SQL condition name	128
Longest SQL descriptor name	128
Longest SQL label	128
Longest statement name	128
Longest unqualified alias name	128
Longest unqualified array type name	128
Longest unqualified column name	128
Longest unqualified constraint name	128
Longest unqualified distinct type name	128
Longest unqualified function name	128
Longest unqualified index name	128
Longest unqualified nodegroup name	10
Longest unqualified package name	10
Longest package version-id	64
Longest unqualified procedure name	128
Longest unqualified sequence name	128
Longest unqualified specific name	128
Longest unqualified SQL parameter name	128
Longest unqualified SQL variable name	128
Longest unqualified system column name	10
Longest unqualified system object name	10
Longest unqualified system schema name	10
Longest unqualified table and view name	128
Longest unqualified trigger name	128
Longest unqualified variable name	128
Longest unqualified XSR object name	128

19. As an application requester, the system can send an authorization name of up to 255 bytes.

SQL path

The *SQL path* is an ordered list of schema names. The database manager uses the path to resolve the schema name for unqualified distinct type names (both built-in types and distinct types), function names, and procedure names that appear in any context other than as the main object of an ALTER, CREATE, DROP, COMMENT, LABEL, GRANT or REVOKE statement.

For example, if the SQL path is SMITH, XGRAPHIC, QSYS, QSYS2 and an unqualified distinct type name MYTYPE was specified, database manager looks for MYTYPE first in schema SMITH, then XGRAPHIC, and then QSYS and QSYS2.

The SQL path used is depends on the SQL statement:

- For static SQL statements (except for a CALL *variable* statement), the path used is the value of the SQLPATH parameter on the CRTSQLxxx command. The SQLPATH can also be set using the SET OPTION statement.

  The path stored in programs, modules, service programs, routines, and triggers, is composed entirely of the *system-schema-names* associated with the schema names in the path. If the *system-schema-name* of a schema is renamed, it may be necessary to recreate these objects if they use SQL statements that depend on the path.
- For dynamic SQL statements (and for a CALL *variable* statement), the path used is the value of the CURRENT PATH special register. For more information about the CURRENT PATH special register, see "CURRENT PATH" on page 135.

If the SQL path is not explicitly specified, the SQL path is the system path followed by the authorization ID of the statement.

For more information about the SQL path for dynamic SQL, see "CURRENT PATH" on page 135.

Qualification of unqualified object names

Unqualified object names are implicitly qualified. The rules for qualifying a name differ depending on the type of object that the name identifies.

Unqualified alias, constraint, external program, index, nodegroup, package, sequence, table, trigger, view, and XSR object names

Unqualified alias, constraint, external program, index, nodegroup, package, sequence, table, trigger, view, and XSR object names are implicitly qualified by the *default schema*.

The *default schema* is specified as follows:

- For static SQL statements:
  - If the DFTRDBCOL parameter is specified on the CRTSQLxxx command (or with the SET OPTION statement), the *default schema* is the *schema-name* that is specified for that parameter.
  - In all other cases, the *default schema* is based on the naming convention.
    - For SQL naming, the *default schema* is the authorization identifier of the statement.
    - For system naming, the *default schema* is the job library list (*LIBL).

20. For REXX procedures, the limit is 33.

- For dynamic SQL statements the *default schema* depends on whether a *default schema* has been explicitly specified. The mechanism for explicitly specifying this depends on the interface used to dynamically prepare and execute SQL statements.
  - If a *default schema* is not explicitly specified:
    - For SQL naming, the *default schema* is the run-time authorization identifier.
    - For system naming, the *default schema* is the job library list (*LIBL).
  - The *default schema* is explicitly specified through the following interfaces:

Table 6. Default Schema Interfaces

SQL Interface	Specification
Embedded SQL	DFTRDBCOL parameter and DYNDFTCOL(*YES) on the Create SQL Program (CRTSQLxxx) and Create SQL Package (CRTSQLPKG) commands. The SET OPTION statement can also be used to set the DFTRDBCOL and DYNDFTCOL values. (For more information about CRTSQLxxx commands, see Embedded SQL Programming.)
Run SQL Statements	DFTRDBCOL parameter on the Run SQL Statements (RUNSQLSTM) command. (For more information about the RUNSQLSTM command, see SQL Programming.)
Call Level Interface (CLI) on the server	SQL_ATTR_DEFAULT_LIB or SQL_ATTR_DBC_DEFAULT_LIB environment or connection variables (For more information about CLI, see SQL Call Level Interfaces (ODBC).)
JDBC or SQLJ on the server using IBM Developer Kit for Java	libraries property object (For more information about JDBC and SQLJ, see IBM Developer Kit for Java.)
ODBC on a client using the IBM i Access Family ODBC Driver	SQL Default Schema in ODBC Setup (For more information about ODBC, see System i Access.)
JDBC on a client using the IBM Toolbox for Java	SQL Default Schema in JDBC Setup (For more information about JDBC, see System i Access.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java.)
OLE DB on a client using the IBM i Access Family OLE DB Provider	DefaultCollection in Connection Object Properties (For more information about OLE DB, see System i Access.)
ADO .NET on a client using the IBM i Access Family ADO .NET Provider	DefaultCollection in Connection Object Properties (For more information about ADO .NET, see System i Access.)
All interfaces	SET SCHEMA or QSQCHGDC (Change Dynamic Default Collection) API (For more information about QSQCHGDC, see the File APIs category.)

Unqualified type, variable, function, procedure, and specific names

The qualification of data type (built-in types, distinct types, and array types), variable, function, procedure, and specific names depends on the SQL statement in which the unqualified name appears.

- If an unqualified name is the main object of an ALTER, CREATE, COMMENT, LABEL, DROP, GRANT, or REVOKE statement, the name is implicitly qualified using the same rules as for qualifying unqualified table names (See "Unqualified alias, constraint, external program, index, nodegroup, package, sequence, table, trigger, view, and XSR object names" on page 63).
- Otherwise, the implicit schema name is determined as follows:
  - For distinct type and array type names, database manager searches the SQL path and selects the first schema in the path such that the data type exists in the schema.
  - For variable names, database manager searches the SQL path and selects the first schema in the path such that the schema contains an authorized variable with the same name
  - For procedure names, database manager searches the SQL path and selects the first schema in the path such that the schema contains an authorized procedure with the same name and number of parameters.
  - For function names, database manager uses the SQL path in conjunction with function resolution, as described under "Function resolution" on page 160.
  - For specific names specified for sourced functions, see "CREATE FUNCTION (Sourced)" on page 894.

SQL names and system names: special considerations

The CL command Override Database File (OVRDBF) can be specified to override an SQL or system name with another object name for local data manipulation SQL statements. Overrides are ignored for data definition SQL statements and data manipulation SQL statements executing at a remote relational database.

See the Database file management topic for more information about the override function.

Aliases

An *alias* can be thought of as an alternative name for a table, partition of a table, view, or member of a database file. A table or view in an SQL statement can be referenced by its name or by an alias. An alias can refer to a table, partition of a table, view, or database file member within the same or a remote relational database.

An alias can be used wherever a table or view name can be used, except:

- Do not use an alias name where a new table or view name is expected, such as in the CREATE TABLE or CREATE VIEW statements. For example, if an alias name of PERSONNEL is created, then a subsequent statement such as CREATE TABLE PERSONNEL will cause an error.
- An alias that refers to an individual partition of a table or member of a database file can only be used in a select statement, CREATE INDEX, DELETE, INSERT, MERGE, SELECT INTO, SET variable, UPDATE, or VALUES INTO statement.

Aliases can also help avoid using file overrides. Not only does an alias perform better than an override, but an alias is also a permanent object that only need be created once.

An alias can be created even though the object that the alias refers to does not exist. However, the object must exist when a statement that references the alias is executed. A warning is returned if the object does not exist when the alias is created. An alias cannot refer to another alias.

Statements that use three-part names and refer to distributed data, result in DRDA access to the remote relational database. When an application program uses three-part name aliases for remote objects and DRDA access, the application program must be bound at each location that is specified in the three-part names. Also, each alias needs to be defined at the local site. An alias at a remote site can refer to yet another server as long as a referenced alias eventually refers to a table or view.

The option of referring to a table, partition of a table, view, or database file member by an alias name is not explicitly shown in the syntax diagrams or mentioned in the description of the SQL statements.

A new alias cannot have the same fully-qualified name as an existing table, view, index, file, or alias.

The effect of using an alias in an SQL statement is similar to that of text substitution. The alias, which must be defined before the SQL statement is executed, is replaced at statement preparation time by the qualified base table, partition of a table, view, or database file member name. For example, if PBIRD.SALES is an alias for DSPN014.DIST4_SALES_148, then at statement run time:

```
SELECT * FROM PBIRD.SALES
```

effectively becomes

```
SELECT * FROM DSPN014.DIST4_SALES_148
```

The effect of dropping an alias and recreating it to refer to another table depends on the statement that references the alias.

- SQL Data or SQL Data Change statements that refer to that alias will be implicitly rebound when they are next run.
- Indexes that reference the alias are not affected.
- Materialized query tables or views that reference the alias are not affected.

For syntax toleration of existing DB2 for z/OS applications, SYNONYM can be used in place of ALIAS in the CREATE ALIAS and DROP ALIAS statements.

Authorization IDs and authorization names

An *authorization ID* is a character string that is obtained by the database manager when a connection is established between the database manager and either an application process or a program preparation process. It designates a set of privileges. It may also designate a user or a group of users, but this property is not controlled by the database manager.

After a connection has been established, the authorization ID may be changed using the SET SESSION AUTHORIZATION statement.

Authorization ID's are used by the database manager to provide authorization checking of SQL statements.

An authorization ID applies to every SQL statement. The authorization ID that is used for authorization checking for a static SQL statement depends on the USRPRF value specified on the precompiler command:

- If USRPRF(*OWNER) is specified, or if USRPRF(*NAMING) is specified and SQL naming mode is used, the authorization ID of the statement is the owner of the non-distributed SQL program. For distributed SQL programs, it is the owner of the SQL package.
- If USRPRF(*USER) is specified, or if USRPRF(*NAMING) is specified and system naming mode is used, the authorization ID of the statement is the authorization ID of the user running the non-distributed SQL program. For distributed SQL programs, it is the authorization ID of the user at the current server.

The authorization ID that is used for authorization checking for a dynamic SQL statement also depends on where and how the statement is executed:

- If the statement is prepared and executed from a non-distributed program:
  - If the USRPRF value is *USER and the DYNUSRPRF value is *USER for the program, the authorization ID that applies is the ID of the user running the non-distributed program. This is called the *run-time authorization ID*.
  - If the USRPRF value is *OWNER and the DYNUSRPRF value is *USER for the program, the authorization ID that applies is the ID of the user running the non-distributed program.
  - If the USRPRF value is *OWNER and the DYNUSRPRF value is *OWNER for the program, the authorization ID that applies is the ID of the owner of the non-distributed program.
- If the statement is prepared and executed from a distributed program:
  - If the USRPRF value is *USER and the DYNUSRPRF value is *USER for the SQL package, the authorization ID that applies is the ID of the user running the SQL package at the current server. This is also called the run-time authorization ID.
  - If the USRPRF value is *OWNER and the DYNUSRPRF value is *USER for the SQL package, the authorization ID that applies is the ID of the user running the SQL package at the current server.
  - If the USRPRF value is *OWNER and the DYNUSRPRF value is *OWNER for the SQL package, the authorization ID that applies is the ID of the owner of the SQL package at the current server.
- If the statement is issued interactively, the authorization ID that applies is the ID of the user that issued the Start SQL (STRSQL) command.
- If the statement is executed from the RUNSQLSTM command, the authorization ID that applies is the ID of the user that issued the RUNSQLSTM command.

- If the statement is executed from REXX, the authorization ID that applies is the ID of the user that issued the STRREXPRC command.

On the IBM i operating system, the run-time authorization ID is the user profile of the job.

An *authorization-name* specified in an SQL statement should not be confused with the authorization ID of the statement. An authorization-name is an identifier that is used in GRANT and REVOKE statements to designate a target of the grant or revoke. The premise of a grant of privileges to *X* is that *X* will subsequently be the authorization ID of statements which require those privileges. A group user profile can also be used when checking authority for an SQL statement. For information about group user profiles, see Security Reference.

Example

Assume SMITH is your user ID; then SMITH is the authorization ID when you execute the following statement interactively:

```
GRANT SELECT ON TDEPT TO KEENE
```

SMITH is the authorization ID of the statement. Thus, the authority to execute the statement is checked against SMITH.

KEENE is an authorization-name specified in the statement. KEENE is given the SELECT privilege on SMITH.TDEPT.

Procedure resolution

Given a procedure invocation, DB2 must decide which of the possible procedures with the same name to execute.

- Let *A* be the number of arguments in a procedure invocation.
- Let *P* be the number of parameters in a procedure signature.
- Let *N* be the number of parameters without a default.

Candidate procedures for resolution of a procedure invocation are selected based on the following criteria:

- Each candidate procedure has a matching name and an applicable number of parameters. An applicable number of parameters satisfies the condition $N \leq A \leq P$.
- Each candidate procedure has parameters such that for each named argument in the CALL statement there exists a parameter with a matching name that does not already correspond to a positional (or unnamed) argument.
- Each parameter of a candidate procedure that does not have a corresponding argument in the CALL statement, specified by either position or name, is defined with a default.
- Each candidate procedure from a set of one or more schemas has the EXECUTE privilege associated with the authorization ID of the CALL statement. The authorities of any objects referenced in a default expression are not considered.

In addition, the set of candidate procedures depends on how the procedure name is qualified.

- If the procedure name is unqualified, procedure resolution is done as follows:

  Search all procedures with a schema in the SQL path for candidate procedures. If one or more candidate procedures are found in the schemas of the SQL path, then these candidate procedures are included in the candidate list. If there is a single candidate procedure in the list, resolution is complete. If there are multiple candidate procedures, choose the procedure whose schema is earliest in the SQL path. If there are still multiple candidate procedures, select the candidate procedure with the lowest number of parameters.

  If there are no candidate procedures, an error is returned.
- If the procedure name is qualified, procedure resolution is done as follows:

  Search within the schema specified by the qualifier for candidate procedures. If a single candidate procedure exists, resolution is complete. If there are multiple candidate procedures, choose the candidate procedure with the lowest number of parameters and resolution is complete. If the schema does not exist or there are no authorized candidate procedures, an error is returned.

Example 1: There are six FOO procedures, in four different schemas, registered as follows (note that not all required keywords appear):

```
CREATE PROCEDURE AUGUSTUS.FOO (INT) SPECIFIC FOO_1 ...
CREATE PROCEDURE AUGUSTUS.FOO (DOUBLE, DECIMAL(15, 3)) SPECIFIC FOO_2 ...
CREATE PROCEDURE JULIUS.FOO (INT) SPECIFIC FOO_3 ...
CREATE PROCEDURE JULIUS.FOO (INT, INT, INT) SPECIFIC FOO_4 ...
CREATE PROCEDURE CAESAR.FOO (INT, INT) SPECIFIC FOO_5 ...
CREATE PROCEDURE NERO.FOO (INT,INT) SPECIFIC FOO_6 ...
```

The procedure reference is as follows (where I1 and I2 are INTEGER values):

```
CALL FOO(I1, I2)
```

Assume that the application making this reference has an SQL path established as:

```
"JULIUS", "AUGUSTUS", "CAESAR"
```

Following through the algorithm, the procedure with specific name FOO_6 is eliminated as a candidate, because the schema "NERO" is not included in the SQL path. FOO_1, FOO_3, and FOO_4 are eliminated as candidates, because they have the wrong number of parameters. The remaining candidates are considered in order, as determined by the SQL path. Note that the types of the arguments and parameters are ignored. The parameters of FOO_5 exactly match the arguments in the CALL, but FOO_2 is chosen because "AUGUSTUS" appears before "CAESAR" in the SQL path.

Example 2: The following examples illustrate procedure resolution using named parameters in the CALL statement:

```
CREATE PROCEDURE p1(i1 INT)...
CREATE PROCEDURE p1(i1 INT DEFAULT 0, i2 INT DEFAULT 0)...

CALL p1(i2=>1)
```

Since the argument names are taken into consideration during the candidate selection process, only the second version of p1 will be considered a candidate. Furthermore, it can be successfully called because i1 in this version of p1 is defined with a default, so only specifying i2 on the call to p1 is valid. The procedure will be passed a value of 0 for parameter i1.

```
CREATE PROCEDURE p2(i1 INT, i2 INT DEFAULT 0)...
CREATE PROCEDURE p2(i1 INT DEFAULT 0, i2 INT DEFAULT 0, i3 INT DEFAULT 0)...

CALL p2(i2=>1)
```

One of the criteria for a procedure parameter which does not have a corresponding argument in the CALL statement (specified by either position or name) is that the parameter is defined with a default value. Therefore, the first version of p2 is not considered a candidate since parameter i1 does not have a default defined. The second version of p2 will be selected and the default values for the first and third parameters will be passed.

Data types

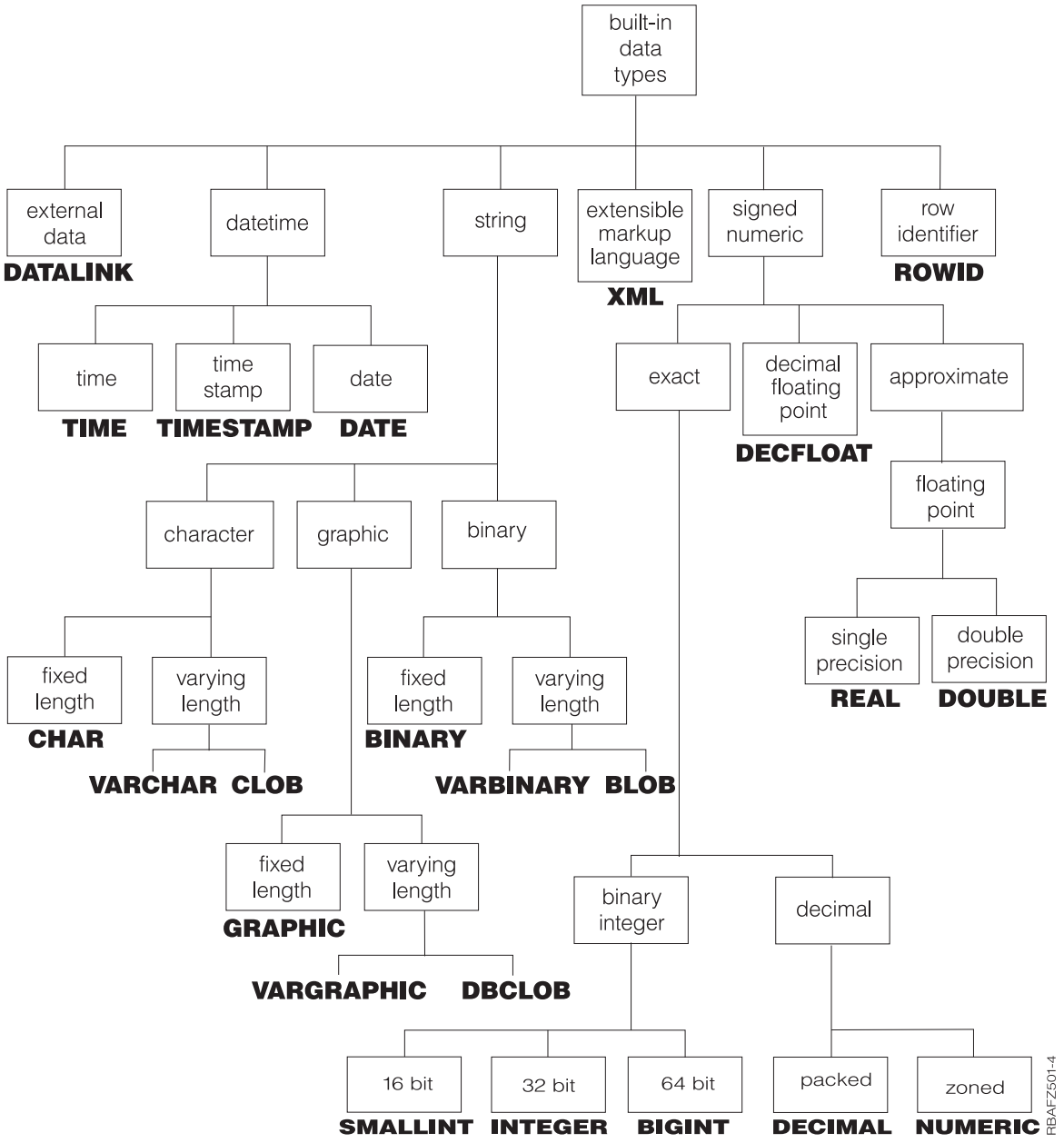
The smallest unit of data that can be manipulated in SQL is called a *value.*

How values are interpreted depends on the *attributes* of their source, which includes the data type, length, precision, scale, and CCSID. The sources of values are:

- Columns
- Constants
- Expressions
- Functions
- Special registers
- Variables (such as host variables, SQL variables, global variables, parameter markers and parameters of routines)

The DB2 relational database products support both built-in data types and user-defined data types. This section describes the built-in data types. For a description of distinct types, see “User-defined types” on page 90.

The following figure illustrates the various built-in data types supported by the DB2 for i program.



For information about specifying the data types of columns, see "CREATE TABLE" on page 982.

Nulls

All data types include the null value. Distinct from all non-null values, the null value is a special value that denotes the absence of a (non-null) value.

Except for grouping operations, a null value is also distinct from another null value. Although all data types include the null value, some sources of values cannot provide the null value. For example, constants and columns that are defined as NOT NULL cannot contain null values, the COUNT and COUNT_BIG

functions cannot return a null value, and ROWID columns cannot store a null value although a null value can be returned for a ROWID column as the result of a query.

Numbers

The numeric data types are binary integer, decimal, decimal floating-point, and floating-point.

The numeric data types are categorized as follows:

- Exact numerics: binary integer and decimal
- Decimal floating-point
- Approximate numerics: floating-point

Binary integer includes small integer, large integer, and big integer. Binary numbers are exact representations of integers. Decimal numbers are exact representations of numbers with a fixed precision and scale. Binary and decimal numbers are considered exact numeric types.

Decimal floating-point numbers can have a precision of 16 or 34. Decimal floating-point supports both exact representations of real numbers and approximations of real numbers and so is not considered either an exact numeric type or an approximate numeric type.

Floating-point includes single precision and double precision. Floating-point numbers are approximations of real numbers and are considered approximate numeric types.

All numbers have a *sign*, a *precision*, and a *scale*. For all numbers except decimal floating-point, if a column or expression is zero, the sign is positive. Decimal floating-point numbers include negative and positive zeros. Decimal floating-point has distinct values for a number and the same number with various exponents (for example: 0.0, 0.00, 0.0E5, 1.0, 1.00, 1.0000). The precision is the total number of digits excluding the sign. The scale is the total number of digits to the right of the decimal point. If there is no decimal point, the scale is zero.

Small integer

A *small integer* is a binary number composed of 2 bytes with a precision of 5 digits. The range of small integers is -32 768 to +32 767.

For small integers, decimal precision and scale are supported by COBOL, RPG, and IBM i system files. For information concerning the precision and scale of binary integers, see the DDS Reference topic.

Large integer

A *large integer* is a binary number composed of 4 bytes with a precision of 10 digits. The range of large integers is -2 147 483 648 to +2 147 483 647.

For large integers, decimal precision and scale are supported by COBOL, RPG, and IBM i system files. For information concerning the precision and scale of binary integers, see the DDS Reference topic.

Big integer

A *big integer* is a binary number composed of 8 bytes with a precision of 19 digits. The range of big integers is -9 223 372 036 854 775 808 to +9 223 372 036 854 775 807.

Decimal

A *decimal* value is a packed decimal or zoned decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 63 digits.

All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is *-n* to *+n*, where the absolute value of *n* is the largest number that can be represented with the applicable precision and scale.

The maximum range is negative 10^{63}+1 to 10^{63} minus 1.

Floating-point

A *single-precision floating-point* number is a 32-bit approximate representation of a real number. The range of magnitude is approximately $1.17549436 \times 10^{-38}$ to $3.40282356 \times 10^{38}$.

A *double-precision floating-point* number is a IEEE 64-bit approximate representation of a real number. The range of magnitude is approximately $2.2250738585072014 \times 10^{-308}$ to $1.7976931348623158 \times 10^{308}$.

Single-precision floating-point is generally accurate to 7 digits of precision. Double-precision floating-point is generally accurate to 15 digits of precision.

Decimal floating-point

A *decimal floating-point* number is an IEEE 754R number with a decimal point. The position of the decimal point is stored in each decimal floating-point value. The maximum precision is 34 digits. The range of a decimal floating-point number is either 16 or 34 digits of precision, and an exponent range of 10^{-383} to 10^{384} or 10^{-6143} to 10^{6144} respectively.

The minimum exponent, E_{min}, for DECFLOAT values is -383 for DECFLOAT(16) and -6143 for DECFLOAT(34). The maximum exponent, E_{max}, for DECFLOAT values is 384 for DECFLOAT(16) and 6144 for DECFLOAT(34).

In addition to the finite numbers, decimal floating-point numbers can also represent the following three special values (see "Decimal floating-point constants" on page 123 for more information):

- Infinity - A value that represents a number whose magnitude is infinitely large.
- Quiet NaN - A value that represents undefined results which does not cause an invalid number warning.

- Signaling NaN - A value that represents undefined results which will cause an invalid number warning if used in any numerical operation.[21]

When a number has one of these special values, its coefficient and exponent are undefined. The sign of an infinity is significant (that is, it is possible to have both positive and negative infinity). The sign of a NaN has no meaning for arithmetic operations.

See Table 112 on page 1496 for more information.

Numeric variables

Small and large binary integer variables can be used in all host languages. Big integer variables can only be used in C, C++, ILE COBOL, and ILE RPG. Floating-point variables can be used in all host languages except RPG/400® and COBOL/400. Decimal variables can be used in all supported host languages. Decimal floating-point variables can only be used in C.

String representations of numeric values

When a decimal, decimal floating-point, or floating-point number is cast to a string (for example, using a CAST specification) the implicit decimal point is replaced by the default decimal separator character in effect when the statement was prepared. When a string is cast to a decimal, decimal floating-point, or floating-point value (for example, using a CAST specification), the default decimal separator character in effect when the statement was prepared is used to interpret the string.

Subnormal numbers and underflow

The decimal floating-point data type has a set of non-zero numbers that fall outside the range of normal decimal floating-point values. These numbers are called subnormal.

Non-zero numbers whose adjusted exponents are less than E_{min} (-6143 for DECFLOAT(34) or -383 for DECFLOAT(16)), are called subnormal numbers. These subnormal numbers are accepted as operands for all operations and may result from any operation. If a result is subnormal before any rounding, the subnormal warning is returned.[22]

For a subnormal result, the minimum value of the exponent becomes E_{min}-(precision - 1), called E_{tiny}, where precision is the precision of the decimal floating-point number. Hence, the smallest value of the exponent E_{tiny} = -6176 for DECFLOAT(34) and -398 for DECFLOAT(16). As the exponent E_{tiny} gets smaller, the number of digits available in the mantissa also decreases. The number of digits available in the mantissa for subnormal numbers is (precision - (- E_{tiny} + E_{min}).

The result will be rounded, if necessary, to ensure that the exponent is no smaller than E_{tiny}. If, during this rounding, the result becomes inexact, an underflow warning is returned.[22] A subnormal result does not always return the underflow warning but will always return the subnormal warning.

When a number underflows to zero during a calculation, its exponent will be E_{tiny}. The maximum value of the exponent is unaffected.

21. The warning is only returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

22. The warning is only returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

The maximum value of the exponent for subnormal numbers is the same as the minimum value of the exponent which can arise during operations that do not result in subnormal numbers. This occurs where the length of the coefficient in decimal digits is equal to the precision.

Character strings

A *character string* is a sequence of bytes. The length of the string is the number of bytes in the sequence. If the length is zero, the value is called the *empty string*. The empty string should not be confused with the null value.

Fixed-length character strings

When fixed-length character string distinct types, columns, and variables are defined, the length attribute is specified and all values have the same length. For a fixed-length character string, the length attribute must be between 1 through 32766 inclusive. See Appendix A, "SQL limits," on page 1493 for more information.

Varying-length character strings

The types of varying-length character strings are:

- VARCHAR
- CLOB

A *Character Large OBject* (CLOB) column is useful for storing large amounts of character data, such as documents written using a single character set.

Distinct types, columns, and variables all have length attributes. When varying-length character-string distinct types, columns, and variables are defined, the maximum length is specified and this becomes the length attribute. Actual values may have a smaller length. For a varying-length character string, the length attribute must be between 1 through 32 740 inclusive. For a CLOB string, the length attribute must be between 1 through 2 147 483 647 inclusive. See Appendix A, "SQL limits," on page 1493 for more information.

For the restrictions that apply to the use of long varying-length strings, see "Limitations on use of strings" on page 81.

Character-string variables

- Fixed-length character-string variables can be used in all host languages except REXX and Java. (In C or C++, fixed-length character-string variables are limited to a length of 1.)
- VARCHAR varying-length character-string variables can be used in C, C++, COBOL, PL/I, REXX, and RPG:
  - In PL/I, REXX, and ILE RPG, there is a varying-length character-string data type.
  - In COBOL, C, and C++ varying-length character strings are represented as structures.
  - In C and C++, varying-length character-string variables can also be represented by NUL-terminated strings.
  - In RPG/400, varying-length character-string variables can only be represented by VARCHAR columns included as a result of an externally described data structure.

- CLOB varying-length character-string variables can be defined in all host languages except REXX, RPG/400, and COBOL/400.
  - In ILE RPG, a CLOB varying-length character string is declared using the SQLTYPE keyword.
  - In all other languages, an SQL TYPE IS CLOB clause is used.

Character encoding schemes

Each character string is further defined as one of four types.

Bit data
Data that is not associated with a coded character set and is therefore never converted. The CCSID for bit data is 65535.

Note: Bit data is a form of character data. The pad character is a blank for assignments to bit data; the pad character is X'00' for assignments to binary data. It is recommended that the binary data type be used instead of character for bit data.

SBCS data
Data in which every character is represented by a single byte. Each SBCS data character string has an associated CCSID. If necessary, an SBCS string is converted before it is used in an operation with a character string that has a different CCSID.

Mixed data
Data that may contain a mixture of characters from a single-byte character set (SBCS) and a double-byte character set (DBCS). Each mixed string has an associated CCSID. If necessary, a mixed data character string is converted before an operation with a character string that has a different CCSID. If mixed data contains a DBCS character, it cannot be converted to SBCS data.

Unicode data
Data that contains characters represented by one or more bytes. Each Unicode character string is encoded using UTF-8. The CCSID for UTF-8 is 1208.

The database manager does not recognize subclasses of double-byte characters, and it does not assign any specific meaning to particular double-byte codes. However, if you choose to use mixed data, then two single-byte EBCDIC codes are given special meanings:

- X'0E', the “shift-out” character, is used to mark the beginning of a sequence of double-byte codes.
- X'0F', the “shift-in” character, is used to mark the end of a sequence of double-byte codes.

In order for the database manager to recognize double-byte characters in a mixed data character string, the following condition must be met:

- Within the string, the double-byte characters must be enclosed between paired shift-out and shift-in characters.

  The pairing is detected as the string is read from left to right. The code X'0E' is recognized as a shift out character if X'0F' occurs later; otherwise, it is invalid. The first X'0F' following the X'0E' that is on a double-byte boundary is the paired shift-in character. Any X'0F' that is not on a double-byte boundary is not recognized.

There must be an even number of bytes between the paired characters, and each pair of bytes is considered to be a double-byte character. There can be more than one set of paired shift-out and shift-in characters in the string.

The length of a mixed data character string is its total number of bytes, counting two bytes for each double-byte character and one byte for each shift-out or shift-in character.

When the job CCSID indicates that DBCS is allowed, CREATE TABLE will create character columns as DBCS-Open fields, unless FOR BIT DATA, FOR SBCS DATA, or an SBCS CCSID is specified. The SQL user will see these as character fields, but the system database support will see them as DBCS-Open fields. For a definition of a DBCS-Open field, see the Database programming topic collection.

Graphic strings

A *graphic string* is a sequence of double-byte characters. The length of the string is the number of its characters. Like character strings, graphic strings can be empty.

Fixed-length graphic strings

When fixed-length graphic-string distinct types, columns, and variables are defined, the length attribute is specified and all values have the same length. For a fixed-length graphic string, the length attribute must be between 1 through 16 383 inclusive. See Appendix A, "SQL limits," on page 1493 for more information.

Varying-length graphic strings

- The types of varying-length graphic strings are:
- VARGRAPHIC
- DBCLOB

A *Double-Byte Character Large OBject* (DBCLOB) column is useful for storing large amounts of double-byte character data, such as documents written using a double-byte character set.

Distinct types, columns, and variables all have length attributes. When varying-length graphic-string distinct types, columns, and variables are defined, the maximum length is specified and this becomes the length attribute. Actual values may have a smaller length. For a varying-length graphic string, the length attribute must be between 1 through 16 370 inclusive. For a DBCLOB string, the length attribute must be between 1 through 1 073 741 823 inclusive. See Appendix A, "SQL limits," on page 1493 for more information.

For the restrictions that apply to the use of long varying-length strings, see "Limitations on use of strings" on page 81.

Graphic-string variables

- Fixed-length graphic-string variables can be defined in C, C++, ILE COBOL, and ILE RPG. (In C and C++, fixed-length graphic-string variables are limited to a length of 1.)

  Although fixed-length graphic-string variables cannot be defined in PL/I, COBOL/400, and RPG/400, a character-string variable will be treated like a fixed-length graphic-string variable if it was generated in the source from a GRAPHIC column in the external definition of a file.

- VARGRAPHIC varying-length graphic-string variables can be defined in C, C++, ILE COBOL, REXX, and ILE RPG.
  - In REXX and ILE RPG, there is a varying-length graphic-string data type.
  - In C, C++, and ILE COBOL, varying-length graphic strings are represented as structures.
  - In C and C++, varying-length graphic-string variables can also be represented by NUL-terminated graphic strings.
  - Although varying-length graphic-string variables cannot be defined in PL/I, COBOL/400, and RPG/400, a character-string variable will be treated like a varying-length graphic-string variable if it was generated in the source from a VARGRAPHIC column in the external definition of a file.
- DBCLOB varying-length character-string variables can be defined in all host languages except REXX, RPG/400, and COBOL/400.
  - In ILE RPG, a DBCLOB varying-length character string is declared using the SQLTYPE keyword.
  - In all other languages, an SQL TYPE IS DBCLOB clause is used.

Graphic encoding schemes

Each graphic string is further defined as one of two types.

DBCS data
Data in which every character is represented by a character from the double-byte character set (DBCS) that does not include the shift-out or shift-in characters.

Every DBCS graphic string has a CCSID that identifies a double-byte coded character set. If necessary, a DBCS graphic string is converted before it is used in an operation with a DBCS graphic string that has a different DBCS CCSID.

Unicode data
Data that contains characters represented by two or more bytes. Each Unicode graphic string is encoded using either UCS-2 or UTF-16. UCS-2 is a subset of UTF-16. The CCSID for UCS-2 is 13488. The CCSID for UTF–16 is 1200.

NCHAR, NVARCHAR, and NCLOB are synonyms for Unicode graphic data with a CCSID of 1200.

When graphic-string variables are not explicitly tagged with a CCSID, the associated DBCS CCSID for the job CCSID is used. If no associated DBCS CCSID exists, the variable is tagged with 65535. A graphic-string variable is never implicitly tagged with a UTF-16 or UCS-2 CCSID. See the DECLARE VARIABLE statement for information on how to tag a graphic variable with a CCSID.

Binary strings

A *binary string* is a sequence of bytes. Unlike a character string which usually contains text data, a binary string is used to hold non-traditional data such as pictures. The length of a binary string is the number of bytes in the sequence. A binary string has a CCSID of 65535. Only character strings of FOR BIT DATA are compatible with binary strings.

Fixed-length binary strings

When fixed-length binary-string distinct types, columns, and variables are defined, the length attribute is specified and all values have the same length. For a fixed-length binary string, the length attribute must be between 1 through 32 766 inclusive. See Appendix A, "SQL limits," on page 1493 for more information.

Varying-length binary strings

The types of varying-length binary strings are:

- VARBINARY
- BLOB

A *Binary Large OBject* (BLOB) column is useful for storing large amounts of noncharacter data, such as pictures, voice, and mixed media. Another use is to hold structured data for exploitation by distinct types and user-defined functions.

Distinct types, columns, and variables all have length attributes. When varying-length binary-string distinct types, columns, and variables are defined, the maximum length is specified and this becomes the length attribute. Actual values may have a smaller length. For a varying-length binary string, the length attribute must be between 1 through 32 740 bytes inclusive. For a BLOB string, the length attribute must be between 1 through 2 147 483 647 inclusive. See Appendix A, "SQL limits," on page 1493 for more information.

Binary-string variables

A variable with a binary string type can be defined in all host languages except REXX, RPG/400, and COBOL/400.

- BINARY fixed-length binary-string variables can be defined in all host languages except REXX, RPG/400, and COBOL/400.
  - In ILE RPG, a BINARY fixed-length binary-string variable is declared using the SQLTYPE keyword.
  - In all other languages, an SQL TYPE IS BINARY clause is used.
- VARBINARY varying-length binary-string variables can be defined in all host languages except REXX, RPG/400, and COBOL/400.
  - In ILE RPG, a VARBINARY varying-length binary-string variable is declared using the SQLTYPE keyword.
  - In all other languages, an SQL TYPE IS VARBINARY clause is used.
- BLOB varying-length binary-string variables can be defined in all host languages except REXX, RPG/400, and COBOL/400.
  - In ILE RPG, a BLOB varying-length binary-string variable is declared using the SQLTYPE keyword.
  - In all other languages, an SQL TYPE IS BLOB clause is used.

Large objects

The term *large object* and the generic acronym *LOB* are used to refer to any CLOB, DBCLOB, or BLOB data type.

Manipulating large objects with locators

Since LOB values can be very large, the transfer of these values from the database server to client application program variables can be time consuming. Also,

application programs typically process LOB values a piece at a time, rather than as a whole. For these cases, the application can reference a LOB value via a large object locator (LOB locator). [23]

A *large object locator* or LOB locator is a variable with a value that represents a single LOB value in the database server. LOB locators were developed to provide users with a mechanism by which they could easily manipulate very large objects in application programs without requiring them to store the entire LOB value on the client machine where the application program may be running.

For example, when selecting a LOB value, an application program could select the entire LOB value and place it into an equally large variable (which is acceptable if the application program is going to process the entire LOB value at once), or it could instead select the LOB value into a LOB locator. Then, using the LOB locator, the application program can issue subsequent database operations on the LOB value by supplying the locator value as input. The resulting output of the locator operation, for example the amount of data assigned to a client variable, would then typically be a small subset of the input LOB value.

LOB locators may also represent more than just base values; they can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

```
SUBSTR(lob_value_1 CONCAT lob_value_2 CONCAT lob_value_3, 42, 6000000)
```

For non-locator-based host variables in an application program, when a null value is selected into that host variable, the indicator variable is set to -1, signifying that the value is null. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Since a LOB locator host variable itself can never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value -- the server does not track null values with valid LOB locators.

It is important to understand that a LOB locator represents a value, not a row or location in the database. Once a value is selected into a LOB locator, there is no operation that one can perform on the original row or table that will affect the value which is referenced by the LOB locator. The value associated with a LOB locator is valid until the transaction ends, or until the LOB locator is explicitly freed, whichever comes first.

A LOB locator is only a mechanism used to refer to a LOB value during a transaction; it does not persist beyond the transaction in which it was created. Also, it is not a database type; it is never stored in the database and, as a result, cannot participate in views or check constraints. However, since a locator is a representation of a LOB type, there are SQLTYPEs for LOB locators so that they can be described within an SQLDA structure that is used by FETCH, OPEN, CALL, and EXECUTE statements.

For the restrictions that apply to the use of LOB strings, see “Limitations on use of strings.”

Limitations on use of strings

Some varying-length string data types cannot be referenced in certain contexts.

23. There is no ability within a Java application to distinguish between a CLOB or BLOB that is represented by a LOB locator and one that is not.

The following varying-length string data types cannot be referenced in certain contexts:

- for character strings, any CLOB string
- for graphic strings, any DBCLOB string
- for binary strings, any BLOB string.

Table 7. Contexts for limitations on use of varying-length strings

Context of usage	LOB (CLOB, DBCLOB, or BLOB)
The CREATE INDEX statement	Not allowed
The definition of primary, unique, and foreign keys	Not allowed
Parameters of built-in functions	Some functions that allow varying-length character strings, varying-length graphic strings, or both types of strings as input arguments do not support CLOB or DBCLOB strings, or both as input. See the description of the individual functions in Chapter 3, "Built-in functions," on page 227 for the data types that are allowed as input to each function.

Datetime values

Although datetime values can be used in certain arithmetic and string operations and are compatible with certain strings, they are neither strings nor numbers.

However, strings can represent datetime values; see "String representations of datetime values" on page 83.

Date

A *date* is a three-part value (year, month, and day) designating a point in time under the Gregorian calendar, which is assumed to have been in effect from the year 1 A.D.

The range of the year part is 0001 to 9999.[24] The date formats *JUL, *MDY, *DMY, and *YMD can only represent dates in the range 1940 through 2039. The range of the month part is 1 to 12. The range of the day part is 1 to *x*, where *x* is 28, 29, 30, or 31, depending on the month and year.

The internal representation of a date is a string of 4 bytes that contains an integer. The integer (called the Scaliger number) represents the date.

The length of a DATE column as described in the SQLDA is 6, 8, or 10 bytes, depending on which format is used. These are the appropriate lengths for string representations for the value.

Time

A *time* is a three-part value (hour, minute, and second) designating a time of day using a 24-hour clock.

24. Note that historical dates do not always follow the Gregorian calendar. Dates between 1582-10-04 and 1582-10-15 are accepted as valid dates although they never existed in the Gregorian calendar.

The range of the hour part is 0 to 24, while the range of the minute and second parts is 0 to 59. If the hour is 24, the minute and second specifications are both zero.

The internal representation of a time is a string of 3 bytes. Each byte consists of two packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

The length of a TIME column as described in the SQLDA is 8 bytes, which is the appropriate length for a string representation of the value.

Timestamp

A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a date and time as defined previously, except that the time includes a fractional specification of microseconds.

The internal representation of a timestamp is a string of 10 bytes. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds (the last 3 bytes contain 6 packed digits).

The length of a TIMESTAMP column as described in the SQLDA is 26 bytes, which is the appropriate length for the string representation of the value.

Datetime variables

Character string variables are normally used to contain date, time, and timestamp values.

The ILE RPG and ILE COBOL precompilers support datetime variables.

Date, time, and timestamp variables can also be specified in Java as java.sql.Date, java.sql.Time, and java.sql.Timestamp, respectively.

String representations of datetime values

Values whose data types are DATE, TIME, or TIMESTAMP are represented in an internal form that is transparent to the user of SQL. Dates, times, and timestamps, however, can also be represented by character or Unicode graphic strings.

To be retrieved, a datetime value can be assigned to a string variable. The format of the resulting string will depend on the *default date format* and the *default time format* in effect when the statement was prepared. The default date and time formats are set based on the date format (DATFMT), the date separator (DATSEP), the time format (TIMFMT), and the time separator (TIMSEP) parameters.

When a valid string representation of a datetime value is used in an operation with an internal datetime value, the string representation is converted to the internal form of the date, time, or timestamp before the operation is performed. The *default date format* and *default time format* specifies the date and time format that will be used to interpret the string. If the CCSID of the string represents a foreign encoding scheme (for example, ASCII), it is first converted to the coded character set identified by the default CCSID before the string is converted to the internal form of the datetime value.

The following sections define the valid string representations of datetime values.

Date strings:

A string representation of a date is a character or a Unicode graphic string that starts with a digit and has a length of at least 6 characters. Trailing blanks can be included. Leading zeros can be omitted from the month and day portions when using the IBM SQL standard formats. Each IBM SQL standard format is identified by name and includes an associated abbreviation (for use by the CHAR function). Other formats do not have an abbreviation to be used by the CHAR function. The separators for two-digit year formats are controlled by the date separator (DATSEP) parameter.

Valid string formats for dates are listed in Table 8.

The database manager recognizes the string as a date when it is in one of the following formats:

- In the format specified by the default date format
- In one of the IBM SQL standard date formats
- In the unformatted Julian format

Table 8. Formats for String Representations of Dates

Format Name	Abbreviation	Date Format	Example
International Standards Organization (*ISO)	ISO	'yyyy-mm-dd'	'1987-10-12'
IBM USA standard (*USA)	USA	'mm/dd/yyyy'	'10/12/1987'
IBM European standard (*EUR)	EUR	'dd.mm.yyyy'	'12.10.1987'
Japanese industrial standard Christian era (*JIS)	JIS	'yyyy-mm-dd'	'1987-10-12'
Unformatted Julian	–	'yyyyddd'	'1987285'
Julian (*JUL)	–	'yy/ddd'	'87/285'
Month, day, year (*MDY)	–	'mm/dd/yy'	'10/12/87'
Day, month, year (*DMY)	–	'dd/mm/yy'	'12/10/87'
Year, month, day (*YMD)	–	'yy/mm/dd'	'87/12/10'

The default date format can be specified through the following interfaces:

Table 9. Default Date Format Interfaces

SQL Interface	Specification
Embedded SQL	The DATFMT and DATSEP parameters are specified on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to specify the DATFMT and DATSEP parameters within the source of a program containing embedded SQL. (For more information about CRTSQLxxx commands, see the Embedded SQL programming topic collection.)
Interactive SQL and Run SQL Statements	The DATFMT and DATSEP parameters on the Start SQL (STRSQL) command or by changing the session attributes. The DATFMT and DATSEP parameters on the Run SQL Statements (RUNSQLSTM) command. (For more information about STRSQL and RUNSQLSTM commands, see SQL programming.)

Table 9. Default Date Format Interfaces (continued)

SQL Interface	Specification
Call Level Interface (CLI) on the server	SQL_ATTR_DATE_FMT and SQL_ATTR_DATE_SEP environment or connection variables (For more information about CLI, see SQL Call Level Interfaces (ODBC).)
JDBC or SQLJ on the server using IBM Developer Kit for Java	Date Format and Date Separator connection property (For more information about JDBC and SQLJ, see IBM Developer Kit for Java.)
ODBC on a client using the IBM i Access Family ODBC Driver	Date Format and Date Separator in the Advanced Server Options in ODBC Setup (For more information about ODBC, see System i Access.)
JDBC on a client using the IBM Toolbox for Java	Format in JDBC Setup (For more information about JDBC, see System i Access.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java.)

Time strings:

A string representation of a time is a character or a Unicode graphic string that starts with a digit and has a length of at least 4 characters. Trailing blanks can be included; a leading zero can be omitted from the hour part of the time and seconds can be omitted entirely. If you choose to omit seconds, an implicit specification of 0 seconds is assumed. Thus, 13.30 is equivalent to 13.30.00.

Valid string formats for times are listed in Table 10. Each IBM SQL standard format is identified by name and includes an associated abbreviation (for use by the CHAR function). The other format (*HMS) does not have an abbreviation to be used by the CHAR function. The separator for the *HMS format is controlled by the time separator (TIMSEP) parameter.

The database manager recognizes the string as a time when it is in one of the following formats:

- In the format specified by the default time format
- In one of the IBM SQL standard time formats

Table 10. Formats for String Representations of Times

Format Name	Abbreviation	Time Format	Example
International Standards Organization (*ISO)	ISO	'hh.mm.ss' [25]	'13.30.05'
IBM USA standard (*USA)	USA	'hh:mm AM' (or PM)	'1:30 PM'
IBM European standard (*EUR)	EUR	'hh.mm.ss'	'13.30.05'
Japanese industrial standard Christian era (*JIS)	JIS	'hh:mm:ss'	'13:30:05'
Hours, minutes, seconds (*HMS)	–	'hh:mm:ss'	'13:30:05'

The following additional rules apply to the USA time format:

25. This is an earlier version of the ISO format. JIS can be used to get the current ISO format.

- The hour must not be greater than 12 and cannot be 0 except for the special case of 00:00 AM.
- A single space character exists between the minutes portion of the time of day and the AM or PM.
- The minutes can be omitted entirely. If you choose to omit the minutes, an implicit specification of 0 minutes is assumed.

In the USA format, using the ISO format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

Table 11. USA Time Format

USA Format	24-Hour Clock
12:01 AM through 12:59 AM	00.01.00 through 00.59.00
01:00 AM through 11:59 AM	01:00.00 through 11:59.00
12:00 PM (noon) through 11:59 PM	12:00.00 through 23.59.00
12:00 AM (midnight)	24.00.00
00:00 AM (midnight)	00.00.00

The default time format can be specified through the following interfaces:

Table 12. Default Time Format Interfaces

SQL Interface	Specification
Embedded SQL	The TIMFMT and TIMSEP parameters are specified on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to specify the TIMFMT and TIMSEP parameters within the source of a program containing embedded SQL. (For more information about CRTSQLxxx commands, see the Embedded SQL programming topic collection.)
Interactive SQL and Run SQL Statements	The TIMFMT and TIMSEP parameters on the Start SQL (STRSQL) command or by changing the session attributes. The TIMFMT and TIMSEP parameters on the Run SQL Statements (RUNSQLSTM) command. (For more information about STRSQL and RUNSQLSTM commands, see the SQL programming topic collection.)
Call Level Interface (CLI) on the server	SQL_ATTR_TIME_FMT and SQL_ATTR_TIME_SEP environment or connection variables (For more information about CLI, see the SQL Call Level Interfaces (ODBC) topic collection.)
JDBC or SQLJ on the server using IBM Developer Kit for Java	Time Format and Time Separator connection property object (For more information about JDBC and SQLJ, see the IBM Developer Kit for Java topic collection.)
ODBC on a client using the IBM i Access Family ODBC Driver	Time Format and Time Separator in the Advanced Server Options in ODBC Setup (For more information about ODBC, see the IBM i Access Family topic collection.)
JDBC on a client using the IBM Toolbox for Java	Format in JDBC Setup (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java topic collection.)

Timestamp strings:

A string representation of a timestamp is a character or a Unicode graphic string that starts with a digit and has a length of at least 14 characters.

The complete string representation of a timestamp has one of the following forms:

Table 13. Formats for String Representations of Timestamps

Format Name	Time Format	Example
ISO timestamp	'yyyy-mm-dd hh:mm:ss.nnnnnn'	'1990-03-02 08:30:00.010000'
IBM SQL	'yyyy-mm-dd-hh.mm.ss.nnnnnn'	'1990-03-02-08.30.00.010000'
14–character form	'yyyymmddhhmmss'	'19900302083000'

Trailing blanks can be included. Leading zeros can be omitted from the month, day, hour, minute, and second part of the timestamp when using the timestamp form with separators. Trailing zeros can be truncated or omitted entirely from microseconds. If you choose to omit any digit of the microseconds portion, an implicit specification of 0 is assumed. Thus, *1990-3-2-8.30.00.10* is equivalent to *1990-03-02-08.30.00.100000*.

A timestamp whose time part is 24.00.00.000000 is also accepted.

XML Values

An XML value represents well-formed XML in the form of an XML document, XML content, or an XML sequence.

An XML value that is stored in a table as a value of a column defined with the XML data type must be a well-formed XML document. XML values are processed in an internal representation that is not comparable to any string value including another XML value. The only predicate that can be applied to the XML data type is the IS NULL predicate.

An XML value can be transformed into a serialized string value representing an XML document using the XMLSERIALIZE function. Similarly, a string value that represents an XML document can be transformed into an XML value using the XMLPARSE function. An XML value can be implicitly parsed or serialized when exchanged with application string and binary data types.

The XML data type has no defined maximum length. It does have an effective maximum length when treated as a serialized string value that represents XML which is the same as the limit for LOB data values. Like LOBs, there are also XML locators and XML file reference variables.

Restrictions when using XML values: With a few exceptions, you can use XML values in the same contexts in which you can use other data types. XML values are valid in:

- CAST a parameter marker, XML, or NULL to XML
- XMLCAST a parameter marker, XML, or NULL to XML
- IS NULL predicate
- COUNT and COUNT_BIG aggregate functions
- COALESCE, IFNULL, HEX, LENGTH, CONTAINS, and SCORE scalar functions
- XML scalar functions
- A SELECT list without DISTINCT
- INSERT VALUES clause, UPDATE SET clause, and MERGE
- SET and VALUES INTO
- Procedure parameters
- User-defined function arguments and result
- Trigger correlation variables
- Parameter marker values for a dynamically prepared statement

XML values cannot be used directly in the following places. Where expressions are allowed, an XML value can be used, for example, as the argument of XMLSERIALIZE:

- A SELECT list containing the DISTINCT keyword
- A GROUP BY clause
- An ORDER BY clause
- A subselect of a fullselect that is not UNION ALL
- A basic, quantified, BETWEEN, DISTINCT, IN, or LIKE predicate
- An aggregate function with the DISTINCT keyword
- A primary, unique, or foreign key
- A check constraint
- An index column

No host languages have a built-in data type for the XML data type.

For information on the XML data model and XML values, see SQL XML programming.

Determining the CCSID for XML

XML data can be defined with any EBCDIC single byte or mixed CCSID or a Unicode CCSID of 1208, 1200, or 13488. 65535 is not allowed as a CCSID for XML data. The CCSID can be explicitly specified when defining an XML data type. If it is not explicitly specified, the CCSID will be assigned using the value of the SQL_XML_DATA_CCSID QAQQINI file option. If this value has not been set, the default is 1208 (UTF-8).

The CCSID will be established for XML data types used in SQL schema statements when the statement is run.

XML host variables that do not have a DECLARE VARIABLE that assigns a CCSID will have their CCSID assigned as follows:.

- If it is XML AS DBCLOB, the CCSID will be 1200.
- If it is XML AS CLOB and the SQL_XML_DATA_CCSID QAQQINI value is 1200 or 13488, the CCSID will be 1208.
- Otherwise, the SQL_XML_DATA_CCSID QAQQINI value will be used as the CCSID.

Since all implicit and explicit XMLPARSE functions are performed using UTF-8 (1208) defining data in this CCSID removes the need to convert the data to UTF-8.

DataLink values

A DataLink value is an encapsulated value that contains a logical reference from the database to a file stored outside the database.

The attributes of this encapsulated value are as follows:

link type
The currently supported type of link is a URL (Uniform Resource Locator).

scheme
For URLs, this is a value such as HTTP or FILE. The value, no matter what case it is entered in, is stored in the database in upper case.

file server name
The complete address of the file server. The value, no matter what case it is entered in, is stored in the database in upper case.

file path
The identity of the file within the server. The value is case sensitive and therefore it is not converted to upper case when stored in the database.

access control token
When appropriate, the access token is embedded within the file path. It is generated dynamically and is not a permanent part of the DataLink value that is stored in the database.

comment
Up to 254 bytes of descriptive information. This is intended for application specific uses such as further or alternative identification of the location of the data.

The characters used in a DataLink value are limited to the set defined for a URL. These characters include the uppercase (A through Z) and lower case (a through z) letters, the digits (0 through 9) and a subset of special characters ($, -, _, @, ., &, +, !, *, '', ', (,), =, ;, /, #, ?, :, space, and comma).

The first four attributes are collectively known as the linkage attributes. It is possible for a DataLink value to have only a comment attribute and no linkage attributes. Such a value may even be stored in a column but, of course, no file will be linked to such a column.

It is important to distinguish between these DataLink references to files and the LOB file reference variables described in "References to LOB or XML file reference variables" on page 154. The similarity is that they both contain a representation of a file. However:

- DataLinks are retained in the database and both the links and the data in the linked files can be considered as a natural extension of data in the database.
- File reference variables exist temporarily and they can be considered as an alternative to a host program buffer.

Built-in scalar functions are provided to build a DataLink value (DLVALUE) and to extract the encapsulated values from a DataLink value (DLCOMMENT, DLLINKTYPE, DLURLCOMPLETE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, DLURLSERVER).

Row ID values

A *row ID* is a value that uniquely identifies a row in a table. A column or a variable can have a row ID data type. A ROWID column enables queries to be written that navigate directly to a row in the table.

Each value in a ROWID column must be unique. The database manager maintains the values permanently, even across table reorganizations. When a row is inserted into the table, the database manager generates a value for the ROWID column unless one is supplied. If a value is supplied, it must be a valid row ID value that was previously generated by either DB2 for z/OS or DB2 for i.

The internal representation of a row ID value is transparent to the user. The value is never subject to CCSID conversion because it is considered to contain BIT data. The length attribute of a ROWID column is 40.

User-defined types

A user-defined type is a data type that is defined to the database using a CREATE TYPE statement. There are two types of user-defined type: distinct types and array types.

Distinct types

A *distinct type* is a user-defined data type that shares its internal representation with a built-in data type (its "source type"), but is considered to be a separate and incompatible type for most operations. For example, the semantics for a picture type, a text type, and an audio type that all use the built-in data type BLOB for their internal representation are quite different. A distinct type is created using "CREATE TYPE (Distinct)" on page 1055.

For example, the following statement creates a distinct type named AUDIO:

```
CREATE TYPE AUDIO AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type that is not comparable to a BLOB or to any other type. This inability to compare AUDIO to other data types allows functions to be created specifically for AUDIO and assures that these functions cannot be applied to other data types (such as pictures or text).

The name of a distinct type is qualified with a schema name. The implicit schema name for an unqualified name depends upon the context in which the distinct type appears. If an unqualified distinct type name is used:

- In a CREATE TYPE statement or the object of the DROP, COMMENT, LABEL, GRANT, or REVOKE statement, the database manager uses the normal process of qualification by authorization ID to determine the schema name. For more information about qualification rules, see "Unqualified type, variable, function, procedure, and specific names" on page 65.
- In any other context, the database manager uses the SQL path to determine the schema name. The database manager searches the schemas in the path, in sequence, and selects the first schema that has a distinct type that matches. For a description of the SQL path, see "SQL path" on page 63.

A distinct type does not automatically acquire the functions and operators of its source type, since these may not be meaningful. (For example, the LENGTH function for an AUDIO type might return the length of its object in seconds rather than in bytes.) Instead, distinct types support *strong typing*. Strong typing ensures that only the functions and operators that are explicitly defined for a distinct type can be applied to that distinct type. However, a function or operator of the source type can be applied to the distinct type by creating an appropriate user-defined function. The user-defined function must be sourced on the existing function that has the source type as a parameter. For example, the following series of SQL statements shows how to create a distinct type named MONEY based on data type DECIMAL(9,2), how to define the + operator for the distinct type, and how the operator might be applied to the distinct type:

```
CREATE TYPE MONEY AS DECIMAL(9,2) WITH COMPARISONS
CREATE FUNCTION "+"(MONEY,MONEY)
   RETURNS MONEY
   SOURCE "+"(DECIMAL(9,2),DECIMAL(9,2))
CREATE TABLE SALARY_TABLE
   (SALARY MONEY,
    COMMISSION MONEY)
SELECT "+"(SALARY, COMMISSION) FROM SALARY_TABLE
```

A distinct type is subject to the same restrictions as its source type. For example, a table can only have one ROWID column. Therefore, a table with a ROWID column cannot also have a column with distinct type that is sourced on a row ID.

The comparison operators are automatically generated for distinct types, except for distinct types that are sourced on a DataLink. In addition, the database manager automatically generates functions for a distinct type that support casting from the source type to the distinct type and from the distinct type to the source type. For example, for the AUDIO type created above, these are the generated cast functions:

Name of generated cast function	Parameter list	Returns data type
schema-name.BLOB	schema-name.AUDIO	BLOB
schema-name.AUDIO	BLOB	schema-name.AUDIO

Array types

An *array* is a structure that contains an ordered collection of data elements. All elements in an array have the same data type. The cardinality of the array is equal to the number of elements in the array.

The entire array can be referenced or an individual element of the array can be referenced by its ordinal position in the collection. If *N* is the cardinality of an array, the ordinal position associated with each element is an integer value greater than or equal to 1 and less than or equal to *N*.

An *array type* is a user-defined data type that is defined as an array. An SQL variable or SQL parameter can be defined as a user-defined array data type. Additionally, the result of an invocation of the TRIM_ARRAY function, or the result of a CAST specification, can be a user-defined array data type. An element of a user-defined array type can be referenced anywhere an expression returning the same data type as an element of that array can be used.

An unnamed array type is an array without an associated user-defined data type. The result of an invocation of the ARRAY_AGG aggregate function or an ARRAY constructor is an array without an associated user-defined data type. An element of an array without an associated user-defined array type cannot be directly referenced.

An array value can be empty (cardinality zero), null, or the individual elements in the array can be null or not null. An empty array is different than an array value of null, or an array for which all elements are the null value.

An array value cannot be stored in the database or be returned to an external application other than Java.

Promotion of data types

Data types can be classified into groups of related data types. Within such groups, an order of precedence exists where one data type is considered to precede another data type. This precedence enables the database manager to support the *promotion* of one data type to another data type that appears later in the precedence ordering. For example, the data type CHAR can be promoted to VARCHAR; INTEGER can be promoted to DOUBLE PRECISION; but CLOB is NOT promotable VARCHAR.

The database manager considers the promotion of data types when:

- performing function resolution (see "Function resolution" on page 160)
- casting distinct types (see "Casting between data types" on page 95)
- assigning distinct types to built-in data types (see "Distinct type comparisons" on page 114)

For each data type, Table 14 shows the precedence list (in order) that the database manager uses to determine the data types to which each data type can be promoted. The table indicates that the best choice is the same data type and not promotion to another data type. Note that the table also shows data types that are considered equivalent during the promotion process. For example, CHARACTER and GRAPHIC are considered to be equivalent data types.

Table 14. Data Type Precedence Table

Data Type	Data Type Precedence List (in best-to-worst order)
SMALLINT	SMALLINT, INTEGER, BIGINT, decimal, real, double, DECFLOAT
INTEGER	INTEGER, BIGINT, decimal, real, double, DECFLOAT
BIGINT	BIGINT, decimal, real, double, DECFLOAT
decimal	decimal, real, double, DECFLOAT
real	real, double, DECFLOAT
double	double, DECFLOAT
DECFLOAT	DECFLOAT
CHAR or GRAPHIC	CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC, CLOB or DBCLOB
CLOB or DBCLOB	CLOB or DBCLOB
CHAR FOR BIT DATA	CHAR, VARCHAR, CLOB, BINARY, VARBINARY, BLOB
VARCHAR FOR BIT DATA	VARCHAR, CLOB, VARBINARY, BLOB
BINARY	BINARY, VARBINARY, BLOB, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA
VARBINARY	VARBINARY, BLOB, VARCHAR FOR BIT DATA
BLOB	BLOB
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
DATALINK	DATALINK

Table 14. Data Type Precedence Table (continued)

Data Type	Data Type Precedence List (in best-to-worst order)
ROWID	ROWID
XML	XML
ARRAY	ARRAY
udt	same udt

Note:

The lower case types above are defined as follows:

decimal
= DECIMAL(p,s) or NUMERIC(p,s)

real = REAL or FLOAT(*n*) where *n* is a specification for single precision floating point

double = DOUBLE, DOUBLE PRECISION, FLOAT or FLOAT(*n*) where *n* is a specification for double precision floating point

udt = a user-defined type

Shorter and longer form synonyms of the data types listed are considered to be the same as the synonym listed.

Character and graphic strings are only compatible for Unicode data.

Casting between data types

There are many occasions when a value with a given data type needs to be cast (changed) to a different data type or to the same data type with a different length, precision, or scale.

Data type promotion, as described in "Promotion of data types" on page 93, is one example of when a value with one data type needs to be cast to a new data type. A data type that can be changed to another data type is *castable* from the source data type to the target data type.

The casting of one data type to another can occur implicitly or explicitly. The cast functions or CAST specification (see "CAST specification" on page 185) can be used to explicitly change a data type. The database manager might implicitly cast data types during assignments that involve a distinct type (see "Distinct type assignments" on page 108). In addition, when you create a sourced user-defined function, the data types of the parameters of the source function must be castable to the data types of the function that you are creating (see "CREATE FUNCTION (Sourced)" on page 894).

If truncation occurs when a character or graphic string is cast to another data type, a warning occurs if any non-blank characters are truncated. This truncation behavior is similar to retrieval assignment of character or graphic strings (see "Retrieval assignment:" on page 103).

If truncation occurs when a binary string is cast to another data type, a warning occurs. This truncation behavior is similar to retrieval assignment of binary strings (see "Retrieval assignment" on page 102).

For casts that involve an array type, the source and target data type must both be the same array type.

For casts that involve a distinct type as either the data type to be cast to or from, Table 15 shows the supported casts. For casts between built-in data types, Table 16 on page 96 shows the supported casts.

Table 15. Supported Casts When a Distinct Type is Involved

Data Type ...	**Is Castable to Data Type ...**
Distinct type *DT*	Source data type of distinct type *DT*
Source data type of distinct type *DT*	Distinct type *DT*
Distinct type *DT*	Distinct type *DT*
Data type *A*	Distinct type *DT* where *A* is promotable to the source data type of distinct type *DT* (see "Promotion of data types" on page 93)
INTEGER	Distinct type *DT* if *DT*'s source type is SMALLINT
DOUBLE	Distinct type *DT* if *DT*'s source data type is REAL
VARCHAR	Distinct type *DT* if *DT*'s source data type is CHAR or GRAPHIC
VARGRAPHIC	Distinct type *DT* if *DT*'s source data type is GRAPHIC or CHAR
VARBINARY	Distinct type *DT* if *DT*'s source data type is BINARY

Character and graphic strings are only compatible for Unicode data. Character bit data and graphic strings are not compatible.

Casting between data types

When a distinct type is involved in a cast, a cast function that was generated when the distinct type was created is used. How the database manager chooses the function depends on whether function notation or the CAST specification syntax is used. For details, see "Function resolution" on page 160, and "CAST specification" on page 185. Function resolution is used for both. However, in a CAST specification, when an unqualified distinct type is specified as the target data type, the database manager resolves the schema name of the distinct type and then uses that schema name to locate the cast function.

The following table describes the supported casts between built-in data types.

Table 16. Supported Casts Between Built-In Data Types

Source Data Type ↓ / Target Data Type →	SMALLINT INTEGER BIGINT	DECIMAL NUMERIC	REAL DOUBLE	DECFLOAT	CHAR VARCHAR CLOB	GRAPHIC VARGRAPHIC DBCLOB	BINARY VARBINARY BLOB	DATE	TIME	TIMESTAMP	ROWID	DATALINK	XML
SMALLINT	Y	Y	Y	Y	Y	Y[1]	—	—	—	—	—	—	—
INTEGER	Y	Y	Y	Y	Y	Y[1]	—	—	—	—	—	—	—
BIGINT	Y	Y	Y	Y	Y	Y[1]	—	—	—	—	—	—	—
DECIMAL	Y	Y	Y	Y	Y	Y[1]	—	—	—	—	—	—	—
NUMERIC	Y	Y	Y	Y	Y	Y[1]	—	—	—	—	—	—	—
REAL	Y	Y	Y	Y	Y	Y[1]	—	—	—	—	—	—	—
DOUBLE	Y	Y	Y	Y	Y	Y[1]	—	—	—	—	—	—	—
DECFLOAT	Y	Y	Y	Y	Y	Y[1]	—	—	—	—	—	—	—
CHAR	Y	Y	Y	Y	Y	Y[1]	Y	Y	Y	Y	Y	—	—
VARCHAR	Y	Y	Y	Y	Y	Y[1]	Y	Y	Y	Y	Y	—	—
CLOB	Y	Y	Y	Y	Y	Y[1]	Y	—	—	—	Y	—	—
GRAPHIC	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y	Y	Y[1]	Y[1]	Y[1]	—	—	—
VARGRAPHIC	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y	Y	Y[1]	Y[1]	Y[1]	—	—	—
DBCLOB	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y	Y	—	—	—	—	—	—
BINARY	—	—	—	—	Y	—	Y	—	—	—	—	—	—
VARBINARY	—	—	—	—	Y	—	Y	—	—	—	—	—	—
BLOB	—	—	—	—	Y	—	Y	—	—	—	—	—	—
DATE	—	—	—	—	Y	—	—	Y	—	Y	—	—	—
TIME	—	—	—	—	Y	—	—	—	Y	Y	—	—	—
TIMESTAMP	—	—	—	—	Y	—	—	Y	Y	Y	—	—	—
ROWID	—	—	—	—	Y	—	Y	—	—	—	Y	—	—
DATALINK	—	—	—	—	—	—	—	—	—	—	—	Y	—
XML	—	—	—	—	—	—	—	—	—	—	—	—	Y

Notes:

[1] Conversion is only supported for Unicode graphic. If the other data type is FOR BIT DATA, conversion is not supported.

The following table describes the rules for casting to a data type:

Table 17. Rules for Casting to a Data Type

Target Data Type	Rules
SMALLINT	See "SMALLINT" on page 491.
INTEGER	See "INTEGER or INT" on page 384.
BIGINT	See "BIGINT" on page 269.

Table 17. Rules for Casting to a Data Type (continued)

Target Data Type	Rules
DECIMAL	See "DECIMAL or DEC" on page 326.
NUMERIC	See "ZONED" on page 589.
REAL	See "REAL" on page 453.
DOUBLE	See "DOUBLE_PRECISION or DOUBLE" on page 344.
DECFLOAT	See "DECFLOAT" on page 323
CHAR	See "CHAR" on page 279.
VARCHAR	See the "VARCHAR" on page 531.
CLOB	See "CLOB" on page 287.
GRAPHIC	If the source data type is a character string, see the rules for string assignment to a variable in "Assignments and comparisons" on page 98. Otherwise, see "GRAPHIC" on page 367.
VARGRAPHIC	If the source data type is a character string, see the rules for string assignment to a variable in "Assignments and comparisons" on page 98. Otherwise, see "VARGRAPHIC" on page 547.
DBCLOB	See "DBCLOB" on page 316.
BINARY	See "BINARY" on page 271.
VARBINARY	See "VARBINARY" on page 528.
BLOB	See "BLOB" on page 275.
DATE	See "DATE" on page 307.
TIME	See "TIME" on page 504.
TIMESTAMP	If the source data type is a string, see "TIMESTAMP" on page 505, where one operand is specified. If the source data type is a DATE, the timestamp is composed of the specified date and a time of 00:00:00. If the source data type is a TIME, the timestamp is composed of the CURRENT_DATE and the specified time.
DATALINK	See the rules for DataLink assignments in "Assignments and comparisons" on page 98.
ROWID	See "ROWID" on page 478.
XML	See the rules for XML assignments in "Assignments and comparisons" on page 98.

Assignments and comparisons

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of CALL, INSERT, UPDATE, FETCH, SELECT, SET variable, and VALUES INTO statements. Comparison operations are performed during the execution of statements that include predicates and other language elements such as MAX, MIN, DISTINCT, GROUP BY, and ORDER BY.

The basic rule for both operations is that the data type of the operands involved must be compatible. The compatibility rule also applies to UNION, EXCEPT, INTERSECT, concatenation, CASE expressions, and the CONCAT, VALUE, COALESCE, IFNULL, MIN, and MAX scalar functions. The compatibility matrix is as follows:

Table 18. Data Type Compatibility

Operands	Binary Integer	Decimal Number	Floating Point	Decimal Floating Point	Character String	Graphic String	Binary String	Date	Time	Timestamp	DataLink	Row ID	XML [8]	User-defined Type
Binary Integer	Y	Y	Y	Y	Y	1	—	—	—	—	—	—	—	4
Decimal Number [5]	Y	Y	Y	Y	Y	1	—	—	—	—	—	—	—	4
Floating Point	Y	Y	Y	Y	Y	1	—	—	—	—	—	—	—	4
Decimal Floating-Point	Y	Y	Y	Y	Y	1	—	—	—	—	—	—	—	4
Character String	Y	Y	Y	Y	Y	1	2	3	3	3	—	—	—	4
Graphic String	1	1	1	1	1	Y	—	1 3	1 3	1 3	—	—	—	4
Binary String	—	—	—	—	2	—	Y	—	—	—	—	—	—	4
Date	—	—	—	—	3	1 3	—	Y	—	—	—	—	—	4
Time	—	—	—	—	3	1 3	—	—	Y	—	—	—	—	4
Timestamp	—	—	—	—	3	1 3	—	—	—	Y	—	—	—	4
DataLink	—	—	—	—	—	—	—	—	—	—	6	—	—	4
Row ID	—	—	—	—	—	—	—	—	—	—	—	7	—	4
XML [8]	—	—	—	—	—	—	—	—	—	—	—	—	Y	4
User-defined Type	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Table 18. Data Type Compatibility (continued)

Operands	Binary Integer	Decimal Number	Floating Point	Decimal Floating Point	Character String	Graphic String	Binary String	Date	Time	Timestamp	DataLink	Row ID	XML [8]	User-defined Type

Notes:

1. Only Unicode graphic strings are compatible. Unicode and FOR BIT DATA are not compatible.
2. Character strings, except those with FOR BIT DATA, are not compatible with binary strings. FOR BIT DATA character strings and binary strings are considered compatible and any padding is performed based on the data type of the target. For example, when assigning a FOR BIT DATA column value to a fixed-length binary variable, any necessary padding uses a pad byte of X'00'.
3. The datetime values and strings are not compatible in concatenation or in the CONCAT scalar function.
4. A value with a distinct type is comparable only to a value that is defined with the same distinct type. In general, the database manager supports assignments between a distinct type value and its source data type. For additional information, see "Distinct type assignments" on page 108.

   A value with an array type is comparable only to a value that is defined with the same array type. A value with an array type can be assigned to an array of the same type. For additional information, see "Array type assignments" on page 109.
5. Decimal refers to both packed and zoned decimal.
6. A DataLink operand can only be assigned to another DataLink operand and cannot be compared to any data type.
7. A ROWID operand can only be assigned to another ROWID operand and cannot be compared to any data type.
8. Character and graphic strings can be assigned to XML columns. However, XML cannot be assigned to a character or graphic string column. For comparisons, XML can only be compared using the IS NULL predicate.

A basic rule for assignment operations is that a null value cannot be assigned to:

- a column that cannot contain null values
- a host variable that does not have an associated indicator variable
- a Java host variable that is a primitive type.

See "References to host variables" on page 149 for a discussion of indicator variables.

For any comparison that involves null values, see the description of the comparison operation for information about the specific handling of null values.

Numeric assignments

For numeric assignments, overflow is not allowed.

- When assigning to an exact numeric data type, overflow occurs if any digit of the whole part of the number would be eliminated. If necessary, the fractional part of a number is truncated.
- When assigning to an approximate numeric data type or decimal floating-point number, overflow occurs if the most significant digit of the whole part of the number is eliminated. For floating-point and decimal floating-point numbers, the whole part of the number is the number that would result if the floating-point

or decimal floating-point number were converted to a decimal number with unlimited precision. If necessary, rounding may cause the least significant digits of the number to be eliminated.

For decimal floating-point numbers, truncation of the whole part of the number is allowed and results in infinity with a warning if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

For floating-point numbers, underflow is also not allowed. Underflow occurs for numbers between 1 and -1 if the most significant digit other than zero would be eliminated. For decimal floating point, underflow is allowed and depending on the rounding mode, results in zero or the smallest positive number or the largest negative number that can be represented. A warning is returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

For information about the decimal floating-point rounding mode, see "CURRENT DECFLOAT ROUNDING MODE" on page 133.

An overflow or underflow warning is returned instead of an error if an overflow or underflow occurs on assignment to a host variable with an indicator variable. In this case, the number is not assigned to the host variable and the indicator variable is set to negative 2.

Assignments to integer

When a decimal, floating-point, or decimal floating-point number is assigned to a binary integer column or variable, the fractional part of the number is eliminated. As a result, a number between 1 and -1 is reduced to 0.

Assignments to decimal

When an integer is assigned to a decimal column or variable, the number is first converted to a temporary decimal number and then, if necessary, to the precision and scale of the target. The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer.

When a decimal number is assigned to a decimal column or variable, the number is converted, if necessary, to the precision and the scale of the target. The necessary number of leading zeros is added, and in the fractional part of the decimal number the necessary number of trailing zeros is added, or the necessary number of trailing digits is eliminated.

When a floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number of precision 63 and scale of 63 - (*p*-*s*) where *p* and *s* are the precision and scale of the decimal column or variable. Then, if necessary, the temporary number is truncated to the precision and scale of the target. As a result, a number between 1 and -1 that is less than the smallest positive number or greater than the largest negative number that can be represented in the decimal column or variable is reduced to 0.

When a decimal floating-point number is assigned to a decimal column or variable, the number is rounded to the precision and scale of the decimal column or variable. As a result, a number between 1 and -1 that is less than the smallest positive number or greater than the largest negative number that can be represented in the decimal column or variable is reduced to 0 or rounded to the smallest positive or largest negative value that can be represented in the decimal column or variable, depending on the rounding mode.

Note: Decimal refers to both packed and zoned decimal. A binary integer with scale follows the rules for a assignments to decimal.

Note: When fetching decimal data from a file that was *not* created by an SQL CREATE TABLE statement, a decimal field may contain data that is not valid. In this case, the data will be returned as stored, without any warning or error message being issued. A table that is created by the SQL CREATE TABLE statement does not allow decimal data that is not valid.

Assignments to floating-point

Floating-point numbers are approximations of real numbers. Hence, when an integer, decimal, floating-point, or decimal floating-point number is assigned to a floating-point column or variable, the result may not be identical to the original number. The number is rounded to the precision of the floating-point column or variable using floating-point arithmetic.

Assignments to decimal floating-point

When an integer number is assigned to a decimal floating-point column or variable, the number is first converted to a temporary decimal number and then to a decimal floating-point number. The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer. Rounding may occur when assigning a BIGINT to a DECFLOAT(16) column or variable.

When a decimal number is assigned to a decimal floating-point column or variable, the number is converted to the precision (16 or 34) of the target. Leading zeros are eliminated. Depending on the precision and scale of the decimal number and the precision of the target, the value might be rounded.

When a floating-point number is assigned to a decimal floating-point column or variable, the number is first converted to a temporary string representation of the floating-point number. The string representation of the number is then converted to decimal floating-point.

When a DECFLOAT(16) number is assigned to a DECFLOAT(34) column or variable, the resulting value is identical to the DECFLOAT(16) number.

When a DECFLOAT(34) number is assigned to a DECFLOAT(16) column or variable, the exponent of the source is converted to the corresponding exponent in the result format. The mantissa of the DECFLOAT(34) number is rounded to the precision of the target. For more information about the decimal floating-point rounding mode, see “CURRENT DECFLOAT ROUNDING MODE” on page 133.

Assignments to COBOL and RPG integers

Assignment to COBOL and RPG small or large integer host variables takes into account any scale specified for the host variable. However, assignment to integer host variables uses the full size of the integer. Thus, the value placed in the COBOL data item or RPG field might be larger than the maximum precision specified for the host variable.

Examples:

- In COBOL, assume that COL1 contains a value of 12345. The following SQL statement results in the value 12345 being placed in A, even though A has been defined with only 4 digits:

```
01  A  PIC  S9999  BINARY.
EXEC SQL SELECT COL1
         INTO :A
         FROM TABLEX
END-EXEC.
```

- Notice, however, that the following COBOL statement results in 2345 (and not 12345) being placed in A:

```
MOVE 12345 TO A.
```

Strings to numeric

When a string is assigned to a numeric data type, it is converted to the target numeric data type using the rules for a CAST specification. For more information, see "CAST specification" on page 185.

String assignments

There are two types of string assignments.

- *Storage assignment* is when a value is assigned to a column or a parameter of a function or stored procedure.
- *Retrieval assignment* is when a value is assigned to a variable.[26]

Binary string assignments

The following rules apply when the assignment target is a binary string.

Storage assignment

The basic rule is that the length of a string assigned to a column or parameter of a function or procedure must not be greater than the length attribute of the column or parameter. If the string is longer than the length attribute of that column or parameter, a error is returned. Trailing hexadecimal zeroes (X'00') are normally included in the length of the string. For storage assignments, however, trailing hexadecimal zeroes are not included in the length of the string.

When a string is assigned to a fixed-length binary-string column or parameter and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of hexadecimal zeroes.

Retrieval assignment

The length of a string assigned to a variable can be greater than the length attribute of the variable. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of bytes. When this occurs, an SQLSTATE of '01004' is assigned to the RETURNED_SQLSTATE condition area item in the SQL Diagnostics Area (or the value 'W' is assigned to the SQLWARN1 field of the SQLCA).

When a string is assigned to a fixed-length binary-string variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of hexadecimal zeroes.

26. If assigning to an SQL-variable or SQL-parameter and the standards option is specified, storage assignment rules apply. For information about the standards option, see "Standards compliance" on page xi.

When a string of length *n* is assigned to a varying-length string variable with a maximum length greater than *n*, the bytes after the *n*th byte of the variable are undefined.

Character and graphic string assignments

The following rules apply when the assignment target is a string.

When a datetime data type is involved, see "Datetime assignments" on page 105. For the special considerations that apply when a distinct type is involved in an assignment, especially to a variable, see "Distinct type assignments" on page 108.

Numeric to strings:

When a number is assigned to a string data type, it is converted to the target string data type using the rules for a CAST specification. For more information, see "CAST specification" on page 185.

Storage assignment:

The basic rule is that the length of a string assigned to a column or parameter of a function or procedure must not be greater than the length attribute of the column or parameter. If the string is longer than the length attribute of that column or parameter, an error is returned. Trailing blanks are normally included in the length of the string. For storage assignments, however, trailing blanks are not included in the length of the string.

When a string is assigned to a fixed-length string column or parameter and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte, double-byte, or UTF-16 or UCS-2 blanks.[27] The pad character is always a blank, even for bit data.

Retrieval assignment:

The length of a string assigned to a variable can be greater than the length attribute of the variable. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of characters. When this occurs, an SQLSTATE of '01004' is assigned to the RETURNED_SQLSTATE condition area item in the SQL Diagnostics Area (or the value 'W' is assigned to the SQLWARN1 field of the SQLCA). Furthermore, if an indicator variable is provided, it is set to the original length of the string. If only the NUL-terminator is truncated for a C NUL-terminated host variable and the *NOCNULRQD option was specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement), an SQLSTATE of '01004' is assigned to the RETURNED_SQLSTATE condition area item in the SQL Diagnostics Area (or the value of 'N' is assigned to the SQLWARN1 field of the SQLCA) and a NUL is not placed in the variable.

When a string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte, double-byte, or UTF-16 or UCS-2 blanks.[27] The pad character is always a blank, even for bit data.

27. UTF-16 or UCS-2 defines a blank character at code point X'0020' and X'3000'. The database manager pads with the blank at code point X'0020'. The database manager pads UTF-8 with a blank at code point X'20'

When a string of length *n* is assigned to a varying-length string variable with a maximum length greater than *n*, the characters after the *n*th character of the variable are undefined.

Assignments to mixed strings:

If a string contains mixed data, the assignment rules may require truncation within a sequence of double-byte codes. To prevent the loss of the shift-in character that ends the double-byte sequence, additional characters may be truncated from the end of the string, and a shift-in character added. In the truncated result, there is always an even number of bytes between each shift-out character and its matching shift-in character.

Assignments to C NUL-terminated strings:

When a string of length *n* is assigned to a C NUL-terminated string variable with a length greater than *n+1*:

- If the *CNULRQD option was specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*YES) on the SET OPTION statement), the string is padded on the right with x-n-1 blanks where x is the length of the variable. The padded string is then assigned to the variable and the NUL-terminator is placed in the next character position.
- If the *NOCNULRQD precompiler option was specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement), the string is not padded on the right. The string is assigned to the variable and the NUL-terminator is placed in the next character position.

Conversion rules for assignments

A string assigned to a column, variable, or parameter is first converted, if necessary, to the coded character set of the target. Character conversion is necessary only if all of the following are true:

- The CCSIDs are different.
- Neither CCSID is 65535.
- The string is neither null nor empty.
- Conversion between the two CCSIDs is required. For more information, see "Coded character sets and CCSIDs" on page 36.

An error occurs if:

- Conversion between the pair of CCSIDs is not defined. For more information, see "Coded character sets and CCSIDs" on page 36.
- A character of the string cannot be converted, and the operation is assignment to a column or assignment to a host variable without an indicator variable. For example, a double-byte character (DBCS) cannot be converted to a column or host variable with a single-byte character (SBCS) CCSID.

A warning occurs if:

- A character of the string is converted to the substitution character.
- A character of the string cannot be converted, and the operation is assignment to a host variable with an indicator variable. For example, a DBCS character cannot be converted to a host variable with an SBCS CCSID. In this case, the string is not assigned to the host variable and the indicator variable is set to -2.

Datetime assignments

A value assigned to a DATE column, a DATE variable, or a DATE parameter must be a date or a valid string representation of a date. A date can be assigned only to a DATE column, a string column, a DATE variable, or a string variable. A value assigned to a TIME column, a TIME variable, or a TIME parameter must be a time or a valid string representation of a time. A time can be assigned only to a TIME column, a string column, a TIME variable, or a string variable. A value assigned to a TIMESTAMP column, a TIMESTAMP variable, or a TIMESTAMP parameter must be a timestamp or a valid string representation of a timestamp. A timestamp can be assigned only to a TIMESTAMP column, a string column, a TIMESTAMP variable, or a string variable.

When a datetime value is assigned to a string variable or column, it is converted to its string representation. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target varies depending on the format of the string representation. If the length of the target is greater than required, it is padded on the right with blanks. If the length of the target is less than required, the result depends on the type of datetime value involved and on the type of target.

- If the target is a string column, truncation is not allowed. The following rules apply:

  DATE

  – The length attribute of the column must be at least 10 if the date format is *ISO, USA, *EUR, or *JIS. If the date format is *YMD, *MDY, or *DMY, the length attribute of the column must be at least 8. If the date format is *JUL, the length of the variable must be at least 6.

  TIME

  – The length attribute of the column must be at least 8.

  TIMESTAMP

  – The length attribute of the column must be at least 26.

- When the target is a variable, the following rules apply:

  DATE

  – The length of the variable must be at least 10 if the date format is *ISO, *USA, *EUR, or *JIS. If the date format is *YMD, *MDY, or *DMY, the length of the variable must be at least 8. If the date format is *JUL, the length of the variable must be at least 6.

  TIME

  – If the *USA format is used, the length of the variable must not be less than 8. This format does not include seconds.

  – If the *ISO, *EUR, *JIS, or *HMS time format is used, the length of the variable must not be less than 5. If the length is 5, 6, or 7, the seconds part of the time is omitted from the result, and SQLWARN1 is set to 'W'. In this case, the seconds part of the time is assigned to the indicator variable if one is provided, and, if the length is 6 or 7, blank padding occurs so that the value is a valid string representation of a time.

  TIMESTAMP

  – The length of the variable must not be less than 19. If the length is between 19 and 25, the timestamp is truncated like a string, causing the omission of one or more digits of the microsecond part. If the length is 20, the trailing decimal point is replaced by a blank so that the value is a valid string representation of a timestamp.

XML assignments

The following rules apply when the assignment target is XML.

The general rule for XML assignments is that only an XML value can be assigned to an XML column or an XML variable. There are exceptions to this rule as follows:

- **Processing of input XML variables:** This is a special case of the XML assignment rule because the variable is based on a string value. To make the assignment to XML within SQL, the string value is implicitly parsed into an XML value using the setting of the CURRENT IMPLICIT XMLPARSE OPTION special register. This determines whether to preserve or to strip whitespace, unless the variable is an argument of the XMLVALIDATE function which always strips unnecessary whitespace.
- **Assigning strings to input parameter markers of data type XML:** If an input parameter marker has an implicit or explicit data type of XML, the value assigned to that parameter marker could be a character string variable, graphic string variable, or binary string variable. In this case, the string value is implicitly parsed into an XML value using the setting of the CURRENT IMPLICIT XMLPARSE OPTION special register. This determines whether to preserve or to strip whitespace, unless the parameter marker is an argument of the XMLVALIDATE function which always strips unnecessary whitespace.
- **Assigning strings directly to XML columns in data change statements:** If assigning directly to a column of type XML in a data change statement, the assigned expression can also be a character string, a graphic string, or a binary string. In this case, the result of XMLPARSE (DOCUMENT *expression* STRIP WHITESPACE) is assigned to the target column. The supported string data types are defined by the supported arguments for the XMLPARSE function. This exception also applies to SQL parameters of type XML.
- **Assigning XML to strings on retrieval:** If retrieving XML values into variables using a FETCH or SELECT INTO in embedded SQL, the data type of the variable can be CLOB, DBCLOB, or BLOB. The XML value is implicitly serialized to a string encoded in the variable's CCSID. If using other application programming interfaces (such as CLI, JDBC, or .NET), XML values can be retrieved into the character, graphic, or binary string types that are supported by the application programming interface. In all these cases, the XML value is implicitly serialized to a string encoded in the CCSID determined by the QAQQINI file as described in “XML Values” on page 88.

For the FETCH, SELECT INTO, SET, and VALUES INTO statements, character string, graphic string, or binary string values cannot be retrieved into XML variables. For the INSERT, UPDATE, MERGE, SET, VALUES INTO, and CALL statements, values in XML variables cannot be assigned to columns, SQL variables, or SQL parameters of a character, graphic, or binary string data type.

DataLink assignments

The assignment of a value to a DataLink column results in the establishment of a link to a file unless the linkage attributes of the value are empty or the column is defined with NO LINK CONTROL. In cases where a linked value already exists in the column, that file is unlinked. Assigning a null value where a linked value already exists also unlinks the file associated with the old value.

If the application provides the same data location as already exists in the column, the link is retained. There are two reasons that this might be done:

- the comment is being changed

- if the table is placed in link pending state, the links in the table can be reinstated by providing linkage attributes identical to the ones in the column.

A DataLink value may be assigned to a column by using the DLVALUE scalar function. The DLVALUE scalar function creates a new DataLink value which can then be assigned a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.

When assigning a value to a DataLink column, the following error conditions can occur:

- Data Location (URL) format is invalid
- File server is not registered with this database
- Invalid link type specified
- Invalid length of comment or URL

  Note that the size of a URL parameter or function result is the same on both input or output and is bound by the length of the DataLink column. However, in some cases the URL value returned has an access token attached. In situations where this is possible, the output location must have sufficient storage space for the access token and the length of the DataLink column. Hence, the actual length of the comment and URL in its fully expanded form provided on input should be restricted to accommodate the output storage space. If the restricted length is exceeded, this error is raised.

When the assignment is also creating a link, the following errors can occur:

- File server not currently available.
- File does not exist.
- Referenced file cannot be accessed for linking.
- File already linked to another column.

  Note that this error will be raised even if the link is to a different relational database.

In addition, when the assignment removes an existing link, the following errors can occur:

- File server not currently available.
- File with referential integrity control is not in a correct state according to the DB2 DataLinks File Manager.

A DataLink value may be retrieved from the database through the use of scalar functions (such as DLLINKTYPE and DLURLPATH). The results of these scalar functions can then be assigned to variables.

Note that usually no attempt is made to access the file server at retrieval time. [28]It is therefore possible that subsequent attempts to access the file server through file system commands might fail.

A warning may be returned when retrieving a DataLink value because the table is in link pending state.

28. It may be necessary to access the file server to determine the prefix name associated with a path. This can be changed at the file server when the mount point of a file system is moved. First access of a file on a server will cause the required values to be retrieved from the file server and cached at the database server for the subsequent retrieval of DataLink values for that file server. An error is returned if the file server cannot be accessed.

Row ID assignments

A row ID value can only be assigned to a column, parameter, or variable with a row ID data type. For the value of the ROWID column, the column must be defined as GENERATED BY DEFAULT or OVERRIDING SYSTEM VALUE must be specified. A unique constraint is implicitly added to every table that has a ROWID column that guarantees that every ROWID value is unique. The value that is specified for the column must be a valid row ID value that was previously generated by DB2 for z/OS or DB2 for i.

Distinct type assignments

The rules that apply to the assignments of distinct types to variables are different than the rules for all other assignments that involve distinct types.

Assignments to variables

The assignment of a distinct type to a variable is based on the source data type of the distinct type. Therefore, the value of a distinct type is assignable to a variable only if the source data type of the distinct type is assignable to the variable.

Example:

Assume that distinct type AGE was created with the following SQL statement:

```
CREATE TYPE AGE AS SMALLINT WITH COMPARISONS
```

When the statement is executed, the following cast functions are also generated:

```
AGE (SMALLINT) RETURNS AGE
AGE (INTEGER) RETURNS AGE
SMALLINT (AGE) RETURNS SMALLINT
```

Next, assume that column STU_AGE was defined in table STUDENTS with distinct type AGE. Now, consider this valid assignment of a student's age to host variable HV_AGE, which has an INTEGER data type:

```
SELECT  STU_AGE INTO :HV_AGE FROM STUDENTS WHERE STU_NUMBER = 200
```

The distinct type value is assignable to the host variable HV_AGE because the source data type of the distinct type (SMALLINT) is assignable to the host variable (INTEGER).

Assignments other than to variables

A distinct type can be either the source or target of an assignment. Assignment is based on whether the data type of the value to be assigned is castable to the data type of the target. “Casting between data types” on page 95 shows which casts are supported when a distinct type is involved. Therefore, a distinct type value can be assigned to any target other than a variable when:

- the target of the assignment has the same distinct type, or
- the distinct type is castable to the data type of the target.

Any value can be assigned to a distinct type when:

- the value to be assigned has the same distinct type as the target, or
- the data type of the assigned value is castable to the target distinct type.

Example:

Assume that the source data type for distinct type AGE is SMALLINT:

```
CREATE TYPE AGE AS SMALLINT WITH COMPARISONS
```

Next, assume that two tables TABLE1 and TABLE2 were created with four identical column descriptions:

```
AGECOL    AGE
SMINTCOL  SMALLINT
INTCOL    INTEGER
DECCOL    DEC(6,2)
```

Using the following SQL statement and substituting various values for X and Y to insert values into various columns of TABLE1 from TABLE2, Table 19 shows whether the assignments are valid.

```
INSERT INTO  TABLE1 (Y) SELECT X FROM TABLE2
```

Table 19. Assessment of various assignments (for example on INSERT)

TABLE2.X	TABLE1.Y	Valid	Reason
AGECOL	AGECOL	Yes	Source and target are same distinct type
SMINTCOL	AGECOL	Yes	SMALLINT can be cast to AGE (because AGE's source type is SMALLINT)
INTCOL	AGECOL	Yes	INTEGER can be cast to AGE (because AGE's source type is SMALLINT)
DECCOL	AGECOL	No	DECIMAL cannot be cast to AGE
AGECOL	SMINTCOL	Yes	AGE can be cast to its source type SMALLINT
AGECOL	INTCOL	No	AGE cannot be cast to INTEGER
AGECOL	DECCOL	No	AGE cannot be cast to DECIMAL

Array type assignments

For array types, the validity of an assignment to an SQL array variable or parameter is determined according to the following rules:

- If the right hand side of the assignment is an SQL array variable or parameter, the TRIM_ARRAY function, or a CAST expression, then its type must be the same type as the SQL array variable or parameter on the left hand side of the assignment.
- If the right hand side of the assignment is an array constructor or the ARRAY_AGG function, then it is implicitly cast to the type of the SQL array variable or parameter on the left hand side.

Assignments to LOB locators

When a LOB locator is used, it can refer to any string data. If a LOB locator is used for the first fetch of a cursor and the cursor is on a remote server, LOB locators must be used for all subsequent fetches unless the *NOOPTLOB precompile option is used.

Numeric comparisons

Numbers are compared algebraically; that is, with regard to sign. For example, –2 is less than +1.

If one number is an integer and the other number is decimal, the comparison is made with a temporary copy of the integer that has been converted to decimal.

When decimal or nonzero scale binary numbers with different scales are compared, the comparison is made with a temporary copy of one of the numbers that has been extended with trailing zeros so that its fractional part has the same number of digits as the other number.

If one number is floating point and the other is integer, decimal, or single-precision floating point, the comparison is made with a temporary copy of the second number converted to a double-precision floating-point number. However, if a single-precision floating-point column is compared to a constant and the constant can be represented by a single-precision floating-point number, the comparison is made with a single-precision form of the constant.

Two floating-point numbers are equal only if the bit configurations of their normalized forms are identical.

If one number is DECFLOAT and the other number is integer, decimal, single precision floating-point, or double precision floating-point, the comparison is made with a temporary copy of the second number converted to DECFLOAT.

If one number is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison.

The DECFLOAT data type supports both positive and negative zero. Positive and negative zero have different binary representations, but the equal (=) predicate will return true for comparisons of positive and negative zero.

The DECFLOAT data type allows for multiple bit representations of the same number. For example, 2.00 and 2.0 are two numbers that are numerically equal but have different bit representations. The = (equal) predicate will return true for a comparison of 2.0 = 2.00. Given that 2.0 = 2.00 is true, 2.0 < 2.00 is false. The behavior that is described here holds true for all comparisons of DECFLOAT values (such as for UNION, SELECT DISTINCT, COUNT DISTINCT, basic predicates, IN predicates, MIN, MAX, and so on.) For example:

```
SELECT 2.0 FROM SYSIBM.SYSDUMMY
UNION
SELECT 2.00 FROM SYSIBM.SYSDUMMY
```

yields one row of data. For this query, the value (2.0 or 2.00) that is returned is arbitrary.

The functions COMPARE_DECFLOAT and TOTALORDER can be used to perform comparisons at a binary level. For example, for a comparison of 2.0<>2.00. With these functions, decimal floating-point values are compared in the following order: -NaN < -sNaN < -Infinity < -0.10 <-0.100 < -0 < 0 < 0.100 < 0.10 < Infinity < sNaN < NaN

The DECFLOAT data type also supports the specification of positive and negative NaN (quiet and signaling), and positive and negative Infinity. From an SQL perspective, infinity = infinity, NaN = NaN, and sNaN = sNaN.

The DECFLOAT data type also supports the specification of positive and negative NaN (quiet and signaling), and positive and negative infinity.

The following rules are the comparison rules for these special values:

- Infinity compares equal only to infinity of the same sign (positive or negative)
- NaN compares equal only to NaN of the same sign (positive or negative)
- sNaN compares equal only to sNaN of the same sign (positive or negative)

When string and numeric data types are compared, the string is converted to the numeric data type with the same precision and scale, and must contain a valid string representation of a number.

String comparisons

There are two different types of string comparisons.

Binary string comparisons

Binary string comparisons always use a collating sequence of *HEX and the corresponding bytes of each string are compared. Additionally, two binary strings are equal only if the lengths of the two strings are identical. If the strings are equal up to the length of the shorter string length, the shorter string is considered less than the longer string even when the remaining bytes in the longer string are hexadecimal zeros. Note that binary strings cannot be compared to character strings unless the character string is cast to a binary string.

Character and graphic string comparisons

Character and Unicode graphic string comparisons use the collating sequence in effect when the statement is executed for all SBCS data and the single-byte portion of mixed data. If the collating sequence is *HEX, the corresponding bytes of each string are compared. For all other collating sequences, the corresponding bytes of the weighted value of each string are compared.

If the strings have different lengths, a temporary copy of the shorter string is padded on the right with blanks before comparison. The padding makes each string the same length. The pad character is always a blank, regardless of the collating sequence. For bit data, the pad character is also a blank. For DBCS graphic data, the pad character is a DBCS blank (x'4040'). For Unicode graphic data, the pad character is a UTF-16 blank. [29]

Two strings are equal if any of the following are true:

- Both strings are empty.
- A *HEX collating sequence is used and all corresponding bytes are equal.
- A collating sequence other than *HEX is used and all corresponding bytes of the weighted value are equal.

An empty string is equal to a blank string. The relationship between two unequal strings is determined by a comparison of the first pair of unequal bytes (or bytes of the weighted value) from the left end of the string. This comparison is made according to the collating sequence in effect when the statement is executed.

29. UTF-16 defines a blank character at code point X'0020' and X'3000'. The database manager pads with the blank at code point X'0020'.

In an application that will run in multiple environments, the same collating sequence (which depends on the CCSIDs of the environments) must be used to ensure identical results. The following table illustrates the differences between EBCDIC, ASCII, and the DB2 LUW default collating sequence for United States English by showing a list that is sorted according to each one.

Table 20. Collating Sequence Differences

ASCII and Unicode	EBCDIC	DB2 LUW Default
0000	@@@@	0000
9999	co-op	9999
@@@@	coop	@@@@
COOP	piano forte	co-op
PIANO-FORTE	piano-forte	COOP
co-op	COOP	coop
coop	PIANO-FORTE	piano forte
piano forte	0000	PIANO-FORTE
piano-forte	9999	piano-forte

Two varying-length strings with different lengths are equal if they differ only in the number of trailing blanks. In operations that select one value from a set of such values, the value selected is arbitrary. The operations that can involve such an arbitrary selection are DISTINCT, MAX, MIN, UNION, EXCEPT, INTERSECT, and references to a grouping column. See “group-by-clause” on page 653 for more information about the arbitrary selection involved in references to a grouping column.

Conversion rules for comparison:

When two strings are compared, one of the strings is first converted, if necessary, to the coded character set of the other string. Character conversion is necessary only if all of the following are true:

- The CCSIDs of the two strings are different.
- Neither CCSID is 65535.
- The string selected for conversion is neither null nor empty.
- Conversion between the two CCSIDs is required. For more information, see “Coded character sets and CCSIDs” on page 36.

If two strings with different encoding schemes are compared, any necessary conversion applies to the string as follows:

Table 21. Selecting the Resulting Encoding Scheme for Character Conversion

	Second Operand			
First Operand	SBCS Data	DBCS Data	Mixed Data	Unicode Graphic Data
SBCS Data	see below	second	second	second
DBCS Data	first	see below	second	second
Mixed Data	first	first	see below	second
Unicode Graphic Data	first	first	first	see below

Otherwise, the string selected for conversion depends on the type of each operand. The following table shows which operand is selected for conversion, given the operand types:

Table 22. Selecting the Operand for Character Conversion

First Operand	Second Operand				
	Column Value	**Derived Value**	**Special Register**	**Constant**	**Variable**
Column Value	second	second	second	second	second
Derived Value	first	second	second	second	second
Special Register	first	first	second	second	second
Constant	first	first	first	second	second
Variable	first	first	first	first	second

A variable that contains data in a foreign encoding scheme is always effectively converted to the native encoding scheme before it is used in any operation. The above rules are based on the assumption that this conversion has already occurred.

An error is returned if a character of the string cannot be converted or if the conversion between the pair of CCSIDs is not defined. For more information, see "Coded character sets and CCSIDs" on page 36. A warning occurs if a character of the string is converted to the substitution character.

Datetime comparisons

A DATE, TIME, or TIMESTAMP value can be compared either with another value of the same data type or with a string representation of that data type. All comparisons are chronological, which means the farther a point in time is from January 1, 0001, the *greater* the value of that point in time.

Comparisons involving TIME values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds are implied. The time 24:00:00 compares greater than the time 00:00:00.

Comparisons involving TIMESTAMP values are chronological without regard to representations that might be considered equivalent. Thus, the following predicate is true:

```
TIMESTAMP('1990-02-23-00.00.00') > '1990-02-22-24.00.00'
```

XML comparisons

The XML data type cannot be compared to any data type, including the XML data type.

DataLink comparisons

A DATALINK operand cannot be directly compared to any data type. The DLCOMMENT, DLLINKTYPE, DLURLCOMPLETE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, and DLURLSERVER scalar functions can be used to extract character string values from a datalink which can then be compared to other strings.

Row ID comparisons

A ROWID operand cannot be directly compared to any data type. To compare the bit representation of a ROWID, first cast the ROWID to a character string.

Distinct type comparisons

A value with a distinct type can be compared only to another value with exactly the same distinct type.

For example, assume that distinct type YOUTH and table CAMP_DB2_ROSTER table were created with the following SQL statements:

```
CREATE TYPE YOUTH AS INTEGER WITH COMPARISONS

CREATE TABLE CAMP_DB2_ROSTER
  ( NAME                VARCHAR(20),
    ATTENDEE_NUMBER     INTEGER NOT NULL,
    AGE                 YOUTH,
    HIGH_SCHOOL_LEVEL   YOUTH)
```

The following comparison is valid because AGE and HIGH_SCHOOL_LEVEL have the same distinct type:

```
SELECT * FROM CAMP_DB2_ROSTER
  WHERE AGE > HIGH_SCHOOL_LEVEL
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER                  ***INCORRECT***
  WHERE AGE > ATTENDEE_NUMBER
```

However, AGE can be compared to ATTENDEE_NUMBER by using a cast function or CAST specification to cast between the distinct type and the source type. All of the following comparisons are valid:

```
SELECT * FROM CAMP_DB2_ROSTER
  WHERE AGE > YOUTH(ATTENDEE_NUMBER)

SELECT * FROM CAMP_DB2_ROSTER
  WHERE AGE > CAST( ATTENDEE_NUMBER AS YOUTH)

SELECT * FROM CAMP_DB2_ROSTER
  WHERE INTEGER(AGE) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB2_ROSTER
  WHERE CAST(AGE AS INTEGER) > ATTENDEE_NUMBER
```

Rules for result data types

The data types of a result are determined by rules which are applied to the operands in an operation. This section explains those rules.

These rules apply to:

- Corresponding columns in UNION, UNION ALL, EXCEPT, or INTERSECT operations
- Result expressions of a CASE expression
- Arguments of the scalar functions COALESCE, IFNULL, MAX, MIN, and VALUE
- Expression values of the IN list of an IN predicate

For the result data type of expressions that involve the operators /, *, + and -, see "With arithmetic operators" on page 166. For the result data type of expressions that involve the CONCAT operator, see "With the concatenation operator" on page 170.

The data type of the result is determined by the data type of the operands. The data types of the first two operands determine an intermediate result data type, this data type and the data type of the next operand determine a new intermediate result data type, and so on. The last intermediate result data type and the data type of the last operand determine the data type of the result. For each pair of data types, the result data type is determined by the sequential application of the rules summarized in the tables that follow.

If neither operand column allows nulls, the result does not allow nulls. Otherwise, the result allows nulls.

If the data type and attributes of any operand column are not the same as those of the result, the operand column values are converted to conform to the data type and attributes of the result. The conversion operation is exactly the same as if the values were assigned to the result. For example,

- If one operand column is CHAR(10), and the other operand column is CHAR(5), the result is CHAR(10), and the values derived from the CHAR(5) column are padded on the right with five blanks.
- If the whole part of a number cannot be preserved then an error is returned.

Numeric operands

Numeric types are compatible with other numeric and character-string and graphic-string data types.

If one operand column is...	And the other operand is...	The data type of the result column is...
SMALLINT	SMALLINT or String	SMALLINT
INTEGER	SMALLINT, INTEGER, or String	INTEGER
BIGINT	SMALLINT, INTEGER, BIGINT, or String	BIGINT
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where p = min(mp, x+max(w-x,5)) mp = 31 or 63 (See Note 1)

Rules for result data types

If one operand column is...	And the other operand is...	The data type of the result column is...
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where p = min(mp, x+max(w-x,11)) mp = 31 or 63 (See Note 1)
DECIMAL(w,x)	BIGINT	DECIMAL(p,x) where p = min(mp, x+max(w-x,19)) mp = 31 or 63 (See Note 1)
DECIMAL(w,x)	DECIMAL(y,z) or NUMERIC(y,z,)	DECIMAL(p,s) where p = min(mp, max(x,z)+max(w-x,y-z)) s = max(x,z) mp = 31 or 63 (See Note 1)
DECIMAL(w,x)	String	DECIMAL(w,x)
NUMERIC(w,x)	SMALLINT	NUMERIC(p,x) where p = min(mp, x + max(w-x,5)) mp = 31 or 63 (See Note 1)
NUMERIC(w,x)	INTEGER	NUMERIC(p,x) where p = min(mp, x + max(w-x,11)) mp = 31 or 63 (See Note 1)
NUMERIC(w,x)	BIGINT	NUMERIC(p,x) where p = min(mp, x + max(w-x,19)) mp = 31 or 63 (See Note 1)
NUMERIC(w,x)	NUMERIC(y,z)	NUMERIC(p,s) where p = min(mp, max(x,z) + max(w-x, y-z)) s = max(x,z) mp = 31 or 63 (See Note 1)
NUMERIC(w,x)	String	NUMERIC(w,x)
NONZERO SCALE BINARY	NONZERO SCALE BINARY	NONZERO SCALE BINARY (If either operand is nonzero scale binary, both operands must be binary with the same scale.)
REAL	REAL	REAL
REAL	SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, or String	DOUBLE
DOUBLE	SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE, or String	DOUBLE
DECFLOAT(n)	REAL, DOUBLE, INTEGER, or SMALLINT	DECFLOAT(n)
DECFLOAT(n)	DECIMAL(p<=16,s) or NUMERIC(p<=16,s)	DECFLOAT(n)
DECFLOAT(n)	BIGINT, DECIMAL(p>16,s), or NUMERIC(p>16,s)	DECFLOAT(34)
DECFLOAT(n)	DECFLOAT(m)	DECFLOAT(max(n,m))
DECFLOAT(n)	String	DECFLOAT(34)

If one operand column is...	And the other operand is...	The data type of the result column is...

Notes:

1. The value of mp is 63 if:
   - either w or y is greater than 31, or
   - a value of 63 was specified for the maximum precision on the DECRESULT parameter of the CRTSQLxxx command, RUNSQLSTM command, or SET OPTION statement

   Otherwise, the value of mp is 31.

Character and graphic string operands

Character and graphic strings are compatible with other character and graphic strings when there is a defined conversion between their corresponding CCSIDs. A character string and a graphic string are compatible if the encoding scheme of the graphic-string data type is Unicode and the character-string data type is not bit data.

If one operand column is...	And the other operand is...	The data type of the result column is...
CHAR(x)	CHAR(y)	CHAR(z) where z = max(x,y)
GRAPHIC(x)	CHAR(y) or GRAPHIC(y)	GRAPHIC(z) where z = max(x,y)
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where z = max(x,y)
VARCHAR(x)	GRAPHIC(y)	VARGRAPHIC(z) where z = max(x,y)
VARGRAPHIC(x)	CHAR(y), VARCHAR(y), GRAPHIC(y), or VARGRAPHIC(y)	VARGRAPHIC(z) where z = max(x,y)
CLOB(x)	CHAR(y), VARCHAR(y), or CLOB(y)	CLOB(z) where z = max(x,y)
CLOB(x)	GRAPHIC(y) or VARGRAPHIC(y)	DBCLOB(z) where z = max(x,y)
DBCLOB(x)	CHAR(y), VARCHAR(y), CLOB(y), GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)	DBCLOB(z) where z = max(x,y)

When one of the operands has a CCSID of 1208, the length (x or y value) of every operand that does not have a CCSID of 1208 is doubled for determining the result length.

The CCSID of the result graphic string will be derived based on the "Conversion rules for operations that combine strings" on page 120.

Binary string operands

Binary strings are compatible only with other binary strings or character strings FOR BIT DATA. Other data types can be treated as a binary-string data type by

using the BINARY, VARBINARY, or BLOB scalar functions to cast the data type to a binary string.

If one operand column is...	And the other operand is...	The data type of the result column is...
BINARY(x)	BINARY(y) or CHAR(y) FOR BIT DATA	BINARY(z) where z = max(x,y)
VARBINARY(x)	BINARY(y), VARBINARY(y), CHAR(y) FOR BIT DATA, or VARCHAR(y) FOR BIT DATA	VARBINARY(z) where z = max(x,y)
VARCHAR(x) FOR BIT DATA	BINARY(y)	VARBINARY(z) where z = max(x,y)
BLOB(x)	BINARY(y), VARBINARY(y), BLOB(y), CHAR(y) FOR BIT DATA, or VARCHAR(y) FOR BIT DATA	BLOB(z) where z = max(x,y)

Datetime operands

A DATE type is compatible with another DATE type or any character or Unicode graphic string expression that contains a valid string representation of a date. A string representation must not be a CLOB. The data type of the result is DATE.

A TIME type is compatible with another TIME type or any character or Unicode graphic string expression that contains a valid string representation of a time. A string representation must not be a CLOB. The data type of the result is TIME.

A TIMESTAMP type is compatible with another TIMESTAMP type or any character or Unicode graphic string expression that contains a valid string representation of a timestamp. A string representation must not be a CLOB. The data type of the result is TIMESTAMP.

If one operand column is...	And the other operand is...	The data type of the result column is...
DATE	DATE, CHAR(y), VARCHAR(y), GRAPHIC(y), or VARGRAPHIC(y)	DATE
TIME	TIME, CHAR(y), VARCHAR(y), GRAPHIC(y), or VARGRAPHIC(y)	TIME
TIMESTAMP	TIMESTAMP, CHAR(y), VARCHAR(y), GRAPHIC(y), or VARGRAPHIC(y)	TIMESTAMP

DataLink operands

A DataLink is compatible with another DataLink. However, DataLinks with NO LINK CONTROL are only compatible with other DataLinks with NO LINK CONTROL; DataLinks with FILE LINK CONTROL READ PERMISSION FS are only compatible with other DataLinks with FILE LINK CONTROL READ PERMISSION FS; and DataLinks with FILE LINK CONTROL READ PERMISSION DB are only compatible with other DataLinks with FILE LINK CONTROL READ PERMISSION DB. The data type of the result is DATALINK. The length of the result DATALINK is the largest length of all the data types.

If one operand column is...	And the other operand is...	The data type of the result column is...
DATALINK(x)	DATALINK(y)	DATALINK(z) where z = max(x,y)

ROWID operands

A ROWID is compatible with another ROWID. The data type of the result is ROWID.

XML operands

The XML data type is compatible only with another XML data type. The data type of the result is XML.

The result CCSID is the value from the SQL_XML_DATA_CCSID QAQQINI option setting as described in “XML Values” on page 88.

User-defined type operands

A user-defined type is compatible only with the same user-defined type. The data type of the result is the user-defined type.

If one operand column is...	And the other operand is...	The data type of the result column is...
User-defined type	User-defined type	User-defined type

Conversion rules for operations that combine strings

The operations that combine strings are concatenation, UNION, UNION ALL, EXCEPT, and INTERSECT. (These rules also apply to the MAX, MIN, VALUE, COALESCE, IFNULL, and CONCAT scalar functions and CASE expressions.) In each case, the CCSID of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the coded character set identified by that CCSID.

The CCSID of the result is determined by the CCSIDs of the operands. The CCSIDs of the first two operands determine an intermediate result CCSID, this CCSID and the CCSID of the next operand determine a new intermediate result CCSID, and so on. The last intermediate result CCSID and the CCSID of the last operand determine the CCSID of the result string or column. For each pair of CCSIDs, the result CCSID is determined by the sequential application of the following rules:

- If the CCSIDs are equal, the result is that CCSID.
- If either CCSID is 65535, the result is 65535.[30]
- If one CCSID denotes data in an encoding scheme different from the other CCSID, the result is determined by the following table:

Table 23. Selecting the Encoding Scheme of the Intermediate Result

	Second Operand			
First Operand	**SBCS Data**	**DBCS Data**	**Mixed Data**	**Unicode Graphic Data**
SBCS Data	see below	second	second	second
DBCS Data	first	see below	second	second
Mixed Data	first	first	see below	second
Unicode Graphic Data	first	first	first	see below

- Otherwise, the resulting CCSID is determined by the following tables:

Table 24. Selecting the CCSID of the Intermediate Result

	Second Operand				
First Operand	**Column Value**	**Derived Value**	**Constant**	**Special Register**	**Variable**
Column Value	first	first	first	first	first
Derived Value	second	first	first	first	first
Constant	second	second	first	first	first
Special Register	second	second	first	first	first
Variable	second	second	second	second	first

30. If either operand is a CLOB or DBCLOB, the resulting CCSID is the job default CCSID.

A variable containing data in a foreign encoding scheme is effectively converted to the native encoding scheme before it is used in any operation. The above rules are based on the assumption that this conversion has already occurred.

Note that an intermediate result is considered to be a derived value operand. For example, assume COLA, COLB, and COLC are columns with CCSIDs 37, 278, and 500, respectively. The result CCSID of COLA CONCAT COLB CONCAT COLC is determined as follows:

1. The result CCSID of COLA CONCAT COLB is first determined to be 37 because both operands are columns, so the CCSID of the first operand is chosen.
2. The result CCSID of "intermediate result" CONCAT COLC is determined to be 500, because the first operand is a derived value and the second operand is a column, so the CCSID of the second operand is chosen.

An operand of concatenation, or the result expression of the CASE expression, or the operands of the IN predicate, or the selected argument of the MAX, MIN, VALUE, COALESCE, IFNULL, or CONCAT scalar function is converted, if necessary, to the coded character set of the result string. Each string of an operand of UNION, UNION ALL, EXCEPT, or INTERSECT is converted, if necessary, to the coded character set of the result column. Character conversion is necessary only if all of the following are true:

- The CCSIDs are different.
- Neither CCSID is 65535.
- The string is neither null nor empty.
- Conversion between the two CCSIDs is required. For more information, see "Coded character sets and CCSIDs" on page 36.

An error is returned if a character of the string cannot be converted or if the conversion between the pair of CCSIDs is not defined. For more information, see "Coded character sets and CCSIDs" on page 36. A warning occurs if a character of a string is converted to the substitution character.

Constants

A *constant* (also called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. String constants are further classified as character or graphic. Numeric constants are further classified as integer, floating point, or decimal.

All constants have the attribute NOT NULL. A negative sign in a numeric constant with a value of zero is ignored.

Integer constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point. The data type of an integer constant is large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of a large integer, but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Examples

```
64      -15     +100        32767    720176    12345678901
```

In syntax diagrams, the term *integer* is used for a large integer constant that must not include a sign.

Decimal constants

A *decimal constant* specifies a decimal number as a signed or unsigned number that consists of no more than 63 digits and either includes a decimal point or is not within the range of binary integers.

The precision is the total number of digits (including leading and trailing zeros); the scale is the number of digits to the right of the decimal point (including trailing zeros). If the precision of the decimal constant is greater than the largest decimal precision and the scale is not greater than the largest decimal precision, then leading zeroes to the left of the decimal point are eliminated to reduce the precision to the largest decimal precision.

Examples

```
25.5    1000.    -15.    +37589.3333333333
```

Floating-point constants

A *floating-point constant* specifies a double-precision floating-point number as two numbers separated by an E. The first number can include a sign and a decimal point; the second number can include a sign but not a decimal point. The value of the constant is the product of the first number and the power of 10 specified by the second number; it must be within the range of floating-point numbers. The number of characters in the constant must not exceed 24. Excluding leading zeros, the number of digits in the first number must not exceed 17 and the number of digits in the second must not exceed 3.

Examples

```
15E1    2.E5    2.2E-1     +5.E+2
```

Decimal floating-point constants

A *decimal floating-point constant* specifies a decimal floating-point number as two numbers separated by an E. The first number can include a sign and a decimal point; the second number can include a sign but not a decimal point. The value of the constant is the product of the first number and the power of 10 specified by the second number; it must be within the range of DECFLOAT(34). The number of characters in the constant must not exceed 42. Excluding leading zeros, the number of digits in the first number must not exceed 34 and the number of digits in the second must not exceed 4.

A constant specified as two numbers separated by E is a decimal floating-point constant only if:

- Excluding leading zeros, the number of digits in the first number exceeds 17 (precision).
- The exponent is outside of the range of double floating-point numbers (smaller than -308 or larger than 308).

In addition to numeric constants, the following reserved keywords can be used to specify decimal floating-point special values. These special values are: INFINITY, NAN, and SNAN. INFINITY represents infinity, a number whose magnitude is infinitely large. INFINITY can be preceded by an optional sign. INF can be specified in place of INFINITY. NAN represents Not a Number (NaN) and is sometimes called quiet NaN. It is a value that represents undefined results which does not cause a warning or exception. SNAN represents signaling NaN (sNaN). It is a value that represents undefined results which will cause a warning or exception if used in any operation that is defined in any numerical operation. Both NAN and SNAN can be preceded by an optional sign, but the sign is not significant for arithmetic operations. SNAN can be used in non-numerical operations without causing a warning or exception, for example in the VALUES list of an INSERT or as a constant compared in a predicate.

When one of the special values (INFINITY, INF, NAN, and SNAN) is used in a context where it could be interpreted as an identifier, such as a column name, cast a string constant representing the special value to decimal-floating point.

```
CAST('snan' AS DECFLOAT(34))
CAST('INF' AS DECFLOAT(34))
CAST('Nan' AS DECFLOAT(34))
```

Examples

```
1.8E308    -1.23456789012345678E-2   SNAN     -INFINITY
```

Character-string constants

A *character-string constant* specifies a varying-length character string.

The two forms of character-string constant follow:

- A sequence of characters that starts and ends with a string delimiter. The number of bytes between the string delimiters cannot be greater than 32740. Two consecutive string delimiters are used to represent one string delimiter within the character string. Two consecutive string delimiters that are not contained within a string represent the empty string.
- An X followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. Blanks between the string delimiters are ignored. The number of hexadecimal digits must not exceed 32762. A hexadecimal digit is a

digit or any of the letters A through F (uppercase or lowercase). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. This form of string constant allows you to specify characters that do not have a keyboard representation.

Character-string constants can contain mixed data. If the job CCSID supports mixed data, a character-string constant is classified as mixed data if it includes a DBCS substring. In all other cases, a character-string constant is classified as SBCS data.

The CCSID assigned to the constant is the CCSID of the source containing the constant unless the source is encoded in a foreign encoding scheme (such as ASCII). The data in the variable is converted from the foreign encoding scheme to the default CCSID of the current server. In this case, the CCSID assigned to the constant is the default CCSID of the current server.

The CCSID of the source is determined by the application requester. The CCSID of the source is:

- For STRSQL, the default CCSID of the application requester
- For the RUNSQLSTM or STRREXPRC commands, the CCSID of the specified source file
- For CRTSQLxxx:
  - For static SQL, the CCSID of the source is the CCSID of the source file used on the CRTSQLxxx command.
  - For dynamic SQL, the CCSID of the source is the CCSID of the variable specified on the PREPARE statement, or if a string constant is specified on the PREPARE statement, the default CCSID of the current server.

Character-string constants are used to represent constant datetime values in assignments and comparisons. For more information see “String representations of datetime values” on page 83.

Examples

```
'Peggy'     '14.12.1990'     '32'     'DON''T CHANGE'     ''     X'FFFF'
```

Graphic-string constants

There are two types of graphic-string constants: DBCS and UTF-16 graphic-string constants.

DBCS graphic-string constants

A *graphic-string constant* is a varying-length graphic string. The length of the specified string cannot be greater than 16370. The three forms of DBCS graphic-string constants are:

Context	Graphic String Constant	Empty String	Example
All contexts	G ' SO dbcs-string SI '	G ' SO SI '	G ' SO 元気 SI '
		G ''	
		g ' SO SI '	
		g ''	
PL/I	SO ' dbcs-string ' G SI	SO ' ' G SI	SO ' 元気 ' G SI

RV3F000-1

In the normal form, the SQL delimiters and the G is an SBCS character. The SBCS ' is the EBCDIC apostrophe, X'7D'.

In the PL/I form, the apostrophes and the G are DBCS characters. Two consecutive DBCS string delimiters are used to represent one string delimiter within the string. Note that this PL/I form is only valid for static statements embedded in PL/I programs.

A hexadecimal DBCS graphic constant is also supported. The form of the hexadecimal DBCS graphic constant is:

In the constant GX'ssss', **ssss** represents a string from 0 to 32760 hexadecimal digits. The number of characters between the string delimiters must be an even multiple of 4. Blanks between the string delimiters are ignored. Each group of 4 digits represents a single DBCS graphic character. The hexadecimal for shift-in and shift-out ('0E'X and '0F'X) are not included in the string.

The CCSID assigned to constants is the DBCS CCSID associated with the CCSID of the source unless the source is encoded in a foreign encoding scheme (such as ASCII). In this case, the CCSID assigned to the constant is the DBCS CCSID associated with the default CCSID of the current server when the SQL statement containing the constant is prepared. If there is no DBCS CCSID associated with the CCSID of the source, the CCSID is 65535.

For information about associated DBCS CCSIDs, see the Use DBCS CCSIDs topic in the IBM i Information Center. For information about the CCSID of the source, see Character String Constants.

Unicode graphic-string constants

There are two types of Unicode graphic-string constants: N and UX. The form of the Unicode graphic constant is:

N'ssss'

In the constant, **ssss** is a string of 16370 characters. The characters are converted from the source CCSID to the Unicode CCSID during processing.

The form of the hexadecimal Unicode graphic constant is:

UX'ssss'

In the constant, **ssss** represents a string from 0 to 32760 hexadecimal digits. The number of characters between the string delimiters must be an even multiple of 4. Blanks between the string delimiters are ignored. Each group of 4 or more digits represents a single Unicode graphic character.

The CCSID of a Unicode graphic-string constant is 13488 (UCS-2). If the standards option is specified, the CCSID is 1200 (UTF-16). For information about the standards option, see “Standards compliance” on page xi.

Binary-string constants

A *binary-string constant* specifies a varying-length binary string.

The two forms of a binary-string constant are:

- A BX followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. Blanks between the string delimiters are ignored. The number of hexadecimal digits must not exceed 32740. A hexadecimal digit is a digit or any of the letters A through F (uppercase or lowercase).
- An X followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. Blanks between the string delimiters are ignored. The number of hexadecimal digits must not exceed 32740. A hexadecimal digit is a digit or any of the letters A through F (uppercase or lowercase).

The CCSID assigned to the constant is 65535.

Note that the syntax of the second form of binary string constant is identical to the second form of a character constant. A constant of this form is only treated as a binary string constant if the standards option is specified. For information on the standards option, see “Standards compliance” on page xi.

Examples

```
BX'FFFF'
X'FFFF'
```

Datetime constants

A *datetime constant* specifies a date, time, or timestamp.

Typically, character-string constants are used to represent constant datetime values in assignments and comparisons. For information on string representations of datetime values, see “String representations of datetime values” on page 83. However, the ANSI/ISO SQL standard form of a datetime constant can be used to specifically denote the constant as a *datetime constant* instead of a string constant.

The format for the three ANSI/ISO SQL standard datetime constants are:

- DATE 'yyyy-mm-dd'

  The data type of the value is DATE.
- TIME 'hh:mm:ss'

  The data type of the value is TIME.
- TIMESTAMP 'yyyy-mm-dd hh:mm:ss.nnnnnn'

  The data type of the value is TIMESTAMP.

  Trailing zeros can be truncated or omitted entirely from microseconds. If you choose to omit any digit of the microseconds portion, an implicit specification of 0 is assumed. Thus, 1990-03-02 08:30:00.10 is equivalent to 1990-03-02 08:30:00.100000.

Leading zeros must not be omitted from any part of a standard datetime constant.

Example

```
DATE '2003-09-03'
```

Decimal point

You can specify a *default decimal point*.

The *default decimal point* can be specified:

- To interpret numeric constants
- To determine the decimal point character to use when casting a character string to a number (for example, in the DECFLOAT, DECIMAL, DOUBLE_PRECISION, FLOAT, and REAL scalar functions and the CAST specification)
- To determine the decimal point character to use in the result when casting a number to a string (for example, in the CHAR, VARCHAR, CLOB, GRAPHIC, and VARGRAPHIC scalar functions and the CAST specification)

The default decimal point can be specified through the following interfaces:

Table 25. Default Decimal Point Interfaces

SQL Interface	Specification
Embedded SQL	The *JOB, *PERIOD, *COMMA, or *SYSVAL value in the OPTION parameter is specified on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to specify the DECMPT parameter within the source of a program containing embedded SQL. (For more information about CRTSQLxxx commands, see Embedded SQL programming.)
Interactive SQL and Run SQL Statements	The DECPNT parameter on the Start SQL (STRSQL) command or by changing the session attributes. The DECMPT parameter on the Run SQL Statements (RUNSQLSTM) command. (For more information about STRSQL and RUNSQLSTM commands, see SQL programming.)
Call Level Interface (CLI) on the server	SQL_ATTR_DATE_FMT and SQL_ATTR_DATE_SEP environment or connection variables (For more information about CLI, see SQL Call Level Interfaces (ODBC).)
JDBC or SQLJ on the server using IBM Developer Kit for Java	Decimal Separator connection property (For more information about JDBC and SQLJ, see IBM Developer Kit for Java.)
ODBC on a client using the IBM i Access Family ODBC Driver	Decimal Separator in the Advanced Server Options in ODBC Setup (For more information about ODBC, see System i Access.)
JDBC on a client using the IBM Toolbox for Java	Format in JDBC Setup (For more information about ODBC, see System i Access.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java.)

If the comma is the decimal point, the following rules apply:

- A period will also be allowed as a decimal point.

- A comma intended as a separator of numeric constants in a list must be followed by a space.
- A comma intended as a decimal point must not be followed by a space.

Thus, to specify a decimal constant without a fractional part, the trailing comma must be followed by a non-blank character. The non-blank character can be a separator comma, as in:

```
VALUES(9999999999,, 111)
```

Delimiters

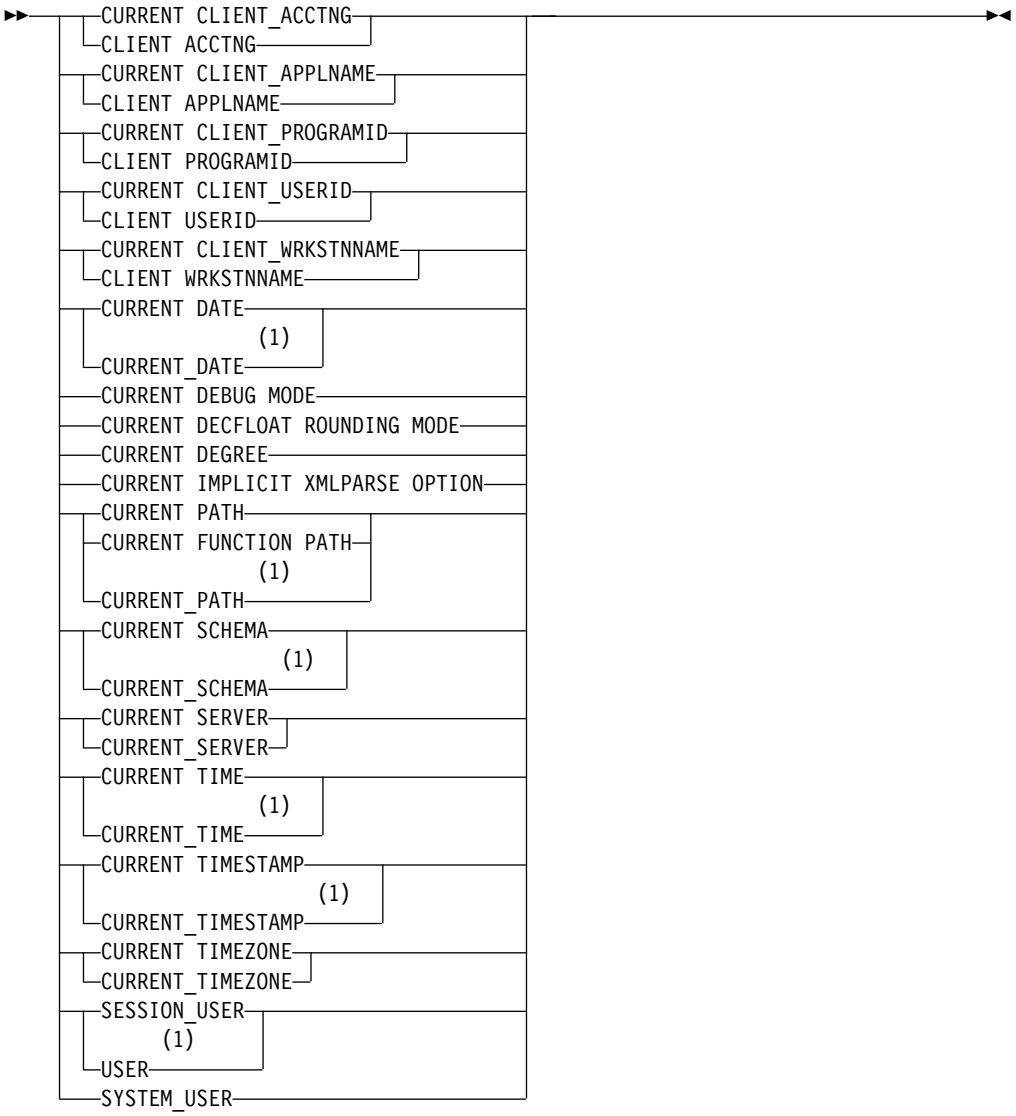
*APOST and *QUOTE are mutually exclusive COBOL precompiler options that name the string delimiter within COBOL statements. *APOST names the apostrophe (') as the string delimiter; *QUOTE names the quotation mark ("). *APOST and *QUOTE are mutually exclusive COBOL precompiler options that play a similar role for SQL statements embedded in COBOL programs. *APOST names the apostrophe (') as the SQL string delimiter; with this option, the quotation mark (") is the SQL escape character. *QUOTE names the quotation mark as the SQL string delimiter; with this option, the apostrophe is the SQL escape character. The values of *APOST and *QUOTE are respectively the same as the values of *APOST and *QUOTE.

In host languages other than COBOL, the usages are fixed. The string delimiter for the host language and for static SQL statements is the apostrophe ('); the SQL escape character is the quotation mark (").

Special registers

A *special register* is a storage area that is defined for an application process by database manager and is used to store information that can be referenced in SQL statements. A reference to a special register is a reference to a value provided by the current server. If the value is a string, its CCSID is a default CCSID of the current server.

The special registers can be referenced as follows:



Notes:

1 The SQL 2003 Core standard uses the form with the underscore.

The value of these special registers cannot be null.

CURRENT CLIENT_ACCTNG

The CURRENT CLIENT_ACCTNG special register specifies a VARCHAR(255) value that contains the value of the accounting string from the client information specified for the current connection.

The default value of this register is the empty string. The value of the accounting string can be changed through these interfaces:

- The Set Client Information (SQLESETI) API can change the client special register.
- The SYSPROC.WLM_SET_CLIENT_INFO procedure can change the client special register.
- In CLI, SQLSetConnectAttr() can be used to set the SQL_ATTR_INFO_ACCTSTR connection attribute.
- In ODBC, SQLSetConnectAttr() can be used to set the ODBC_ATTR_INFO_ACCTSTR connection attribute.
- In JDBC, the setClientInfo connection method can be used to set the ClientAccounting connection property.

Example

Get the current value of the accounting string for this connection

```
VALUES CURRENT CLIENT_ACCTNG
  INTO :ACCT_STRING
```

CURRENT CLIENT_APPLNAME

The CURRENT CLIENT_APPLNAME special register specifies a VARCHAR(255) value that contains the value of the application name from the client information specified for the current connection.

The default value of this register is the empty string. The value of the application name can be changed through these interfaces:

- The Set Client Information (SQLESETI) API can change the client special register.
- The SYSPROC.WLM_SET_CLIENT_INFO procedure can change the client special register.
- In CLI, SQLSetConnectAttr() can be used to set the SQL_ATTR_INFO_APPLNAME connection attribute.
- In ODBC, SQLSetConnectAttr() can be used to set the ODBC_ATTR_INFO_APPLNAME connection attribute.
- In JDBC, the setClientInfo connection method can be used to set the ApplicationName connection property.

Example

Select the departments that are allowed to use the application being used in this connection.

```
SELECT  DEPT
FROM DEPT_APPL_MAP
WHERE APPL_NAME = CURRENT CLIENT_APPLNAME
```

CURRENT CLIENT_PROGRAMID

The CURRENT CLIENT_PROGRAMID special register specifies a VARCHAR(255) value that contains the value of the client program ID from the client information specified for the current connection.

The default value of this register is the empty string. The value of the client program ID can be changed through these interfaces:

- The Set Client Information (SQLESETI) API can change the client special register.
- The SYSPROC.WLM_SET_CLIENT_INFO procedure can change the client special register.
- In CLI, SQLSetConnectAttr() can be used to set the SQL_ATTR_INFO_PROGRAMID connection attribute.
- In ODBC, SQLSetConnectAttr() can be used to set the ODBC_ATTR_INFO_PROGRAMID connection attribute.
- In JDBC, the setClientInfo connection method can be used to set the ClientProgramID connection property.

Example

Get the program ID being used for this connection.

```
VALUES CURRENT CLIENT_PROGRAMID
INTO :PGM_ID
```

CURRENT CLIENT_USERID

The CURRENT CLIENT_USERID special register specifies a VARCHAR(255) value that contains the value of the client user ID from the client information specified for the current connection.

The default value of this register is the empty string. The value of the client user ID can be changed through these interfaces:

- The Set Client Information (SQLESETI) API can change the client special register.
- The SYSPROC.WLM_SET_CLIENT_INFO procedure can change the client special register.
- In CLI, SQLSetConnectAttr() can be used to set the SQL_ATTR_INFO_USERID connection attribute.
- In ODBC, SQLSetConnectAttr() can be used to set the ODBC_ATTR_INFO_USERID connection attribute.
- In JDBC, the setClientInfo connection method can be used to set the ClientUser connection property.

Example

Find out in which department the current client user ID works.

```
SELECT  DEPT
FROM DEPT_USERID_MAP
WHERE USER_ID = CURRENT CLIENT_USERID
```

CURRENT CLIENT_WRKSTNNAME

The CURRENT CLIENT_WRKSTNNAME special register specifies a VARCHAR(255) value that contains the value of the workstation name from the client information specified for the current connection.

The default value of this register is the empty string. The value of the workstation name can be changed through these interfaces:

- The Set Client Information (SQLESETI) API can change the client special register.
- The SYSPROC.WLM_SET_CLIENT_INFO procedure can change the client special register.

- In CLI, SQLSetConnectAttr() can be used to set the SQL_ATTR_INFO_WRKSTNNAME connection attribute.
- In ODBC, SQLSetConnectAttr() can be used to set the ODBC_ATTR_INFO_WRKSTNNAME connection attribute.
- In JDBC, the setClientInfo connection method can be used to set the ClientHostName connection property.

Example

Get the workstation name being used for this connection.

```
VALUES CURRENT CLIENT_WRKSTNNAME
INTO :WS_NAME
```

CURRENT DATE

The CURRENT DATE special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

If this special register is used more than once within a single SQL statement, or used with CURRENT TIME, CURRENT TIMESTAMP, or the CURDATE, CURTIME, or NOW scalar functions within a single statement; all values are based on a single clock reading.[31]

Example

Using the PROJECT table, set the project end date (PRENDATE) of the MA2111 project (PROJNO) to the current date.

```
UPDATE PROJECT
  SET PRENDATE = CURRENT DATE
  WHERE PROJNO = 'MA2111'
```

CURRENT DEBUG MODE

The CURRENT DEBUG MODE special register specifies whether SQL or Java procedures should be created or altered so they can be debugged by the Unified Debugger.

Any explicit specification of the DEBUG MODE or the DBGVIEW option in the SET OPTION statement on the CREATE PROCEDURE or ALTER PROCEDURE statement overrides the value in the CURRENT DEBUG MODE special register. CURRENT DEBUG MODE affects static and dynamic SQL statements. The data type of the register is VARCHAR(8). The valid values include:

DISALLOW
Procedures will be created so they cannot be debugged by the Unified Debugger. When the DEBUG MODE attribute of a procedure is DISALLOW, the procedure can be subsequently altered to change the DEBUG MODE attribute.

ALLOW
Procedures will be created so they can be debugged by the Unified Debugger. When the DEBUG MODE attribute of a procedure is ALLOW, the procedure can be subsequently altered to change the DEBUG MODE attribute.

DISABLE
Procedures will be created so they cannot be debugged by the Unified

31. LOCALDATE can be specified as a synonym for CURRENT_DATE.

Debugger. When the DEBUG MODE attribute of a procedure is DISABLE, the procedure cannot be subsequently altered to change the DEBUG MODE attribute.

The value can be changed by invoking the SET CURRENT DEBUG MODE statement. For details about this statement, see "SET CURRENT DEBUG MODE" on page 1351.

The initial value of CURRENT DEBUG MODE is DISALLOW.

Example

The following statement prevents subsequent creates or alters of SQL or Java procedures from being debuggable:

```
SET CURRENT DEBUG MODE = DISALLOW
```

CURRENT DECFLOAT ROUNDING MODE

The CURRENT DECFLOAT ROUNDING MODE special register specifies the rounding mode that is used when DECFLOAT values are manipulated in dynamically prepared SQL statements.

The data type of the register is VARCHAR(128). The rounding modes supported are:

ROUND_CEILING
Round toward +infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by one (rounded up).

ROUND_DOWN
Round toward zero (truncation). The discarded digits are ignored.

ROUND_FLOOR
Round toward -infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged other than the removal of the discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by one.

ROUND_HALF_DOWN
Round to nearest; if equidistant, round down. If the discarded digits represent greater than half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). Otherwise, the discarded digits are ignored.

ROUND_HALF_EVEN
Round to nearest; if equidistant, round so that the final digit is even. If the discarded digits represent greater than half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise (they represent exactly half), the result coefficient is unaltered if its rightmost digit is even, or incremented by one (rounded up) if its rightmost digit is odd (to make an even digit).

ROUND_HALF_UP
Round to nearest; if equidistant, round up. If the discarded digits represent

greater than or equal to half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). Otherwise, the discarded digits are ignored.

ROUND_UP
Round away from zero. If all of the discarded digits are zero, the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by one (rounded up).

The initial value of CURRENT DECFLOAT ROUNDING MODE in an activation group is established by the first SQL statement that is executed in the activation group.

- If the first SQL statement in an activation group is executed from an SQL program or SQL package, the CURRENT DECFLOAT ROUNDING MODE special register is set to the value of the DECFLTRND parameter.
- Otherwise, the initial value is ROUND_HALF_EVEN.

The DECFLTRND parameter on the CRTSQLxxx command or SET OPTION is used for static SQL statements.

Example

Set the host variable APPL_ROUND (VARCHAR(128)) to the current rounding mode.

```
SELECT CURRENT DECFLOAT ROUNDING MODE
  INTO :APPL_ROUND
  FROM SYSIBM.SYSDUMMY1
```

CURRENT DEGREE

The CURRENT DEGREE special register specifies the degree of I/O or Symmetric MultiProcessing (SMP) parallelism for the execution of queries, index creates, index rebuilds, index maintenance, and reorganizes.

CURRENT DEGREE affects static and dynamic SQL statements. The data type of the register is CHAR(5). The valid values include:

1 No parallel processing is allowed.

2 through 32767
Specifies the degree of parallelism that will be used.

ANY
Specifies that the database manager can choose to use any number of tasks for either I/O or SMP parallel processing.

Use of parallel processing and the number of tasks used is determined based on the number of processors available in the system, this job's share of the amount of active memory available in the pool in which the job is run, and whether the expected elapsed time for the operation is limited by CPU processing or I/O resources. The database manager chooses an implementation that minimizes elapsed time based on the job's share of the memory in the pool.

NONE
No parallel processing is allowed.

MAX
The database manager can choose to use any number of tasks for either I/O or

SMP parallel processing. MAX is similar to ANY except the database manager assumes that all active memory in the pool can be used.

IO Any number of tasks can be used when the database manager chooses to use I/O parallel processing for queries. SMP is not allowed.

The initial value of CURRENT DEGREE is determined by the current degree in effect from the CHGQRYA CL command, PARALLEL_DEGREE parameter in the current query options file (QAQQINI), or the QQRYDEGREE system value.

The value can be changed by invoking the SET CURRENT DEGREE statement. For details about this statement, see "SET CURRENT DEGREE" on page 1356.

Example

The following statement inhibits parallelism:

```
SET CURRENT DEGREE  = '1'
```

CURRENT IMPLICIT XMLPARSE OPTION

The CURRENT IMPLICIT XMLPARSE OPTION special register specifies whitespace handling options that are to be used when serialized XML data is implicitly parsed by the DB2 server without validation.

An implicit non-validating parse operation occurs when an SQL statement is processing an XML host variable or an implicitly or explicitly typed XML parameter marker that is not an argument of the XMLVALIDATE function.

The data type of the register is VARCHAR(128). The supported values are:

STRIP WHITESPACE
Whitespace that is in the document to improve readability is removed. The whitespace characters are: blank, carriage return, line feed, and tab. All boundary whitespace (whitespace between elements) is removed.

PRESERVE WHITESPACE
No whitespace is removed.

The initial value of CURRENT IMPLICIT XMLPARSE OPTION is 'STRIP WHITESPACE'.

Example

Set the CURRENT IMPLICIT XMLPARSE OPTION to PRESERVE WHITESPACE.

```
SET CURRENT IMPLICIT XMLPARSE OPTION = PRESERVE WHITESPACE
```

CURRENT PATH

The CURRENT PATH special register specifies the SQL path used to resolve unqualified distinct type names, function names, and procedure names in dynamically prepared SQL statements.

It is also used to resolve unqualified procedure names that are specified as variables in SQL CALL statements (CALL *variable*). The data type is VARCHAR(8843).

The CURRENT PATH special register contains the value of the SQL path, which is a list of one or more schema names. Each schema name is enclosed in delimiters

and separated from the following schema by a comma (any delimiters within the string are repeated as they are in any delimited identifier). The delimiters and commas are included in the length of the special register. The maximum number of schema names in the path is 268.

If a schema name has a different system name, the schema name is returned in the CURRENT PATH special register even if the system name was explicitly specified in the SET PATH statement.

For information about when the SQL path is used to resolve unqualified names in both dynamic and static SQL statements and the effect of its value, see "Unqualified type, variable, function, procedure, and specific names" on page 65.

The initial value of the CURRENT PATH special register in an activation group is established by the first SQL statement that is executed.

- If the first SQL statement in an activation group is executed from an SQL program or SQL package and the SQLPATH parameter was specified on the CRTSQLxxx command, the path is the value specified in the SQLPATH parameter. The SQLPATH value can also be specified using the SET OPTION statement.
- Otherwise,
  - For SQL naming, "QSYS", "QSYS2", "SYSPROC", "SYSIBMADM", "*the value of the run-time authorization ID of the statement*" .
  - For system naming, "*LIBL".

The value of the special register can be changed by executing the SET PATH statement. For details about this statement, see "SET PATH" on page 1387. For portability across the platforms, it is recommended that a SET PATH statement be issued at the beginning of an application.

Example

Set the special register so that schema SMITH is searched before schemas QSYS and QSYS2 (SYSTEM PATH).

```
SET CURRENT PATH SMITH, SYSTEM PATH
```

CURRENT SCHEMA

The CURRENT SCHEMA special register specifies a VARCHAR(128) value that identifies the schema name used to qualify unqualified database object references where applicable in dynamically prepared SQL statements.

CURRENT SCHEMA is not used to qualify names in programs where the DYNDFTCOL has been specified. If DYNDFTCOL is specified in a program, its schema name is used instead of the CURRENT SCHEMA schema name.[32]

The initial value of CURRENT SCHEMA is the authorization ID of the current session user.

The value of the special register can be changed by executing the SET SCHEMA statement. For more information, see "SET SCHEMA" on page 1393.

32. For compatibility with DB2 for z/OS, the special register CURRENT SQLID is treated as a synonym for CURRENT SCHEMA.

The DFTRDBCOL keyword controls the schema name used to qualify unqualified database object references where applicable for static SQL statements.

Example

Set the schema for object qualification to 'D123'.

```
SET CURRENT SCHEMA = 'D123'
```

CURRENT SERVER

The CURRENT SERVER special register specifies a VARCHAR(18) value that identifies the current application server.

CURRENT SERVER can be changed by the CONNECT (Type 1), CONNECT (Type 2), or SET CONNECTION statements, but only under certain conditions. See the description in “CONNECT (Type 1)” on page 836, “CONNECT (Type 2)” on page 842, and “SET CONNECTION” on page 1348.

CURRENT SERVER cannot be specified unless the local relational database is named by adding the entry to the relational database directory using the ADDRDBDIRE or WRKRDBDIRE command.

Example

Set the host variable APPL_SERVE (VARCHAR(18)) to the name of the current server.

```
SELECT CURRENT SERVER
  INTO :APPL_SERVE
  FROM SYSIBM.SYSDUMMY1
```

CURRENT TIME

The CURRENT TIME special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

If this special register is used more than once within a single SQL statement, or used with CURRENT DATE, CURRENT TIMESTAMP, or the CURDATE, CURTIME, or NOW scalar functions within a single statement; all values are based on a single clock reading.[33]

Example

Using the CL_SCHED table, select all the classes (CLASS_CODE) that start (STARTING) later today. Today's classes have a value of 3 in the DAY column.

```
SELECT CLASS_CODE FROM CL_SCHED
  WHERE STARTING > CURRENT TIME  AND DAY = 3
```

CURRENT TIMESTAMP

The CURRENT TIMESTAMP special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

33. LOCALTIME and LOCALTIME(0) can be specified as a synonyms for CURRENT_TIME.

If this special register is used more than once within a single SQL statement, or used with CURRENT DATE, CURRENT TIME, or the CURDATE, CURTIME, or NOW scalar functions within a single statement; all values are based on a single clock reading.[34]

Example

Insert a row into the IN_TRAY sample table. The value of the RECEIVED column should be a timestamp that indicates when the row was inserted. The values for the other three columns come from the host variables SRC (CHAR(8)), SUB (CHAR(64)), and TXT (VARCHAR(200)).

```
INSERT INTO IN_TRAY
  VALUES (CURRENT TIMESTAMP, :SRC, :SUB, :TXT)
```

CURRENT TIMEZONE

The CURRENT TIMEZONE special register specifies the difference between UTC and local time at the current server.

The difference is represented by a time duration (a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds).[35] The number of hours is between -24 and 24 exclusive. Subtracting CURRENT TIMEZONE from a local time converts that local time to UTC.

Example

Using the IN_TRAY table, select all the rows from the table and adjust the value to UTC.

```
SELECT RECEIVED - CURRENT TIMEZONE, SOURCE,
      SUBJECT, NOTE_TEXT FROM IN_TRAY
```

SESSION_USER

The SESSION_USER special register specifies the run-time authorization ID at the current server. The data type of the special register is VARCHAR(128).

The initial value of SESSION_USER for a new connection is the same as the value of the SYSTEM_USER special register.

The value can be changed by executing the SET SESSION AUTHORIZATION statement. For more information, see “SET SESSION AUTHORIZATION” on page 1396.

Example

Select all notes from the IN_TRAY table that the user placed there himself.

```
SELECT * FROM IN_TRAY
  WHERE SOURCE = SESSION_USER
```

SYSTEM_USER

The SYSTEM_USER special register specifies the authorization ID that connected to the current server. The data type of the special register is VARCHAR(128).

34. LOCALTIMESTAMP and LOCALTIMESTAMP(6) can be specified as a synonym for CURRENT_TIMESTAMP.

35. Coordinated Universal Time, formerly known as GMT.

Example

Select all notes from the IN_TRAY table that the user placed there himself.

```
SELECT * FROM IN_TRAY
  WHERE SOURCE = SYSTEM_USER
```

USER

The USER special register specifies the run-time authorization ID at the current server. The data type of the special register is VARCHAR(18).

The initial value of USER for a new connection is the same as the value of the SYSTEM_USER special register.

The value can be changed by executing the SET SESSION AUTHORIZATION statement. For more information, see “SET SESSION AUTHORIZATION” on page 1396.

Example

Select all notes from the IN_TRAY table that the user placed there himself.

```
SELECT * FROM IN_TRAY
  WHERE SOURCE = USER
```

Column names

The meaning of a column name depends on its context.

A column name can be used to:

- Declare the name of a column, as in a CREATE TABLE statement.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
  - In an *aggregate function*, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. Groups and intermediate result tables are explained under Chapter 5, "Queries," on page 627. For example, MAX(SALARY) applies the function MAX to all values of the column SALARY in a group.
  - In a *GROUP BY or ORDER BY clause*, a column name specifies all values in the intermediate result table to which the clause is applied. For example, ORDER BY DEPT orders an intermediate result table by the values of the column DEPT.
  - In an *expression*, a *search condition*, or a *scalar function*, a column name specifies a value for each row or group to which the construct is applied. For example, when the search condition CODE = 20 is applied to some row, the value specified by the column name CODE is the value of the column CODE in that row.
- Provide a column name for an expression to temporarily rename a column, as in the *correlation-clause* of a *table-reference* in a FROM clause, or in the AS clause in the *select-clause*.

Qualified column names

A qualifier for a column name can be a table name, a view name, an alias name, or a correlation name.

Whether a column name can be qualified depends on its context:

- In the COMMENT and LABEL statements, the column name must be qualified.
- Where the column name specifies values of the column, a column name can be qualified.
- In the *assignment-clause* of an UPDATE statement, it may be qualified.
- In the *column-name-list* of an INSERT statement, it may be qualified.
- In all other contexts, a column name must not be qualified.

Where a qualifier is optional it can serve two purposes. See "Column name qualifiers to avoid ambiguity" on page 142 and "Column name qualifiers in correlated references" on page 144 for details.

Correlation names

A *correlation name* can be defined in the FROM clause of a query and after the target *table-name* or *view-name* in an UPDATE or DELETE statement.

For example, the clause shown below establishes Z as a correlation name for X.MYTABLE:

```
FROM X.MYTABLE Z
```

A correlation name is associated with a table or view only within the context in which it is defined. Hence, the same correlation name can be defined for different purposes in different statements, or in different clauses of the same statement.

As a qualifier, a correlation name can be used to avoid ambiguity or to establish a correlated reference. A correlation name can also be used as a shorter name for a table or view. In the example that is shown above, Z might have been used merely to avoid having to enter X.MYTABLE more than once.

If a correlation name is specified for a table or view, any qualified reference to a column of that instance of the table or view must use the correlation name, rather than the table name or view name. For example, the reference to EMPLOYEE.PROJECT in the following example is incorrect, because a correlation name has been specified for EMPLOYEE:

```
FROM EMPLOYEE E                          ***INCORRECT***
  WHERE EMPLOYEE.PROJECT='ABC'
```

The qualified reference to PROJECT should instead use the correlation name, “E”, as shown below:

```
FROM EMPLOYEE E
  WHERE E.PROJECT='ABC'
```

Names specified in a FROM clause are either *exposed* or *non-exposed*. A correlation name is always an exposed name. A table name or view name is said to be *exposed* in that FROM clause if a correlation name is not specified. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE but not for DEPARTMENT, so DEPARTMENT is an exposed name, and EMPLOYEE is not:

```
FROM EMPLOYEE E, DEPARTMENT
```

A table name or view name that is exposed in a FROM clause must not be the same as any other table name or view name exposed in that FROM clause or any correlation name in the FROM clause. The names are compared after qualifying any unqualified table or view names.

The first two FROM clauses shown below are correct, because each one contains no more than one reference to EMPLOYEE that is exposed:

1. Given the FROM clause:

   ```
   FROM EMPLOYEE E1, EMPLOYEE
   ```

   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the second instance of EMPLOYEE in the FROM clause. A qualified reference to the first instance of EMPLOYEE must use the correlation name “E1” (E1.PROJECT).
2. Given the FROM clause:

   ```
   FROM EMPLOYEE, EMPLOYEE E2
   ```

   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the first instance of EMPLOYEE in the FROM clause. A qualified reference to the second instance of EMPLOYEE must use the correlation name “E2” (E2.PROJECT).
3. Given the FROM clause:

   ```
   FROM EMPLOYEE, EMPLOYEE                  ***INCORRECT***
   ```

   the two exposed table names included in this clause (EMPLOYEE and EMPLOYEE) are the same, and this is not allowed.

4. Given the following statement:

```
SELECT *
  FROM EMPLOYEE E1, EMPLOYEE E2            ***INCORRECT***
  WHERE EMPLOYEE.PROJECT='ABC'
```

   the qualified reference EMPLOYEE.PROJECT is incorrect, because both instances of EMPLOYEE in the FROM clause have correlation names. Instead, references to PROJECT must be qualified with either correlation name (E1.PROJECT or E2.PROJECT).

5. Given the FROM clause:

```
FROM EMPLOYEE, X.EMPLOYEE
```

   a reference to a column in the second instance of EMPLOYEE must use X.EMPLOYEE (X.EMPLOYEE.PROJECT). This FROM clause is only valid if the authorization ID of the statement is not X.

A correlation name specified in a FROM clause must not be the same as:

- Any other correlation name in that FROM clause
- Any unqualified table name or view name exposed in the FROM clause
- The second SQL identifier of any qualified table name or view name that is exposed in the FROM clause.

For example, the following FROM clauses are incorrect:

```
FROM EMPLOYEE E, EMPLOYEE E
FROM EMPLOYEE DEPARTMENT, DEPARTMENT           ***INCORRECT***
FROM X.T1, EMPLOYEE T1
```

The following FROM clause is technically correct, though potentially confusing:

```
FROM EMPLOYEE DEPARTMENT, DEPARTMENT EMPLOYEE
```

The use of a correlation name in the FROM clause also allows the option of specifying a list of column names to be associated with the columns of the result table. As with a correlation name, these listed column names become the exposed names of the columns that must be used for references to the columns throughout the query. If a column name list is specified, then the column names of the underlying table become non-exposed.

Given the FROM clause:

```
FROM DEPARTMENT D (NUM,NAME,MGR,ANUM,LOC)
```

a qualified reference such as D.NUM denotes the first column of the DEPARTMENT table that is defined in the table as DEPTNO. A reference to D.DEPTNO using this FROM clause is incorrect since the column name DEPTNO is a non-exposed column name.

If a list of columns is specified, it must consist of as many names as there are columns in the *table-reference*. Each column name must be unique and unqualified.

Column name qualifiers to avoid ambiguity

In the context of a function, a GROUP BY clause, an ORDER BY clause, an expression, or a search condition, a column name refers to values of a column in some target table or view in a DELETE or UPDATE statement or *table-reference* in a FROM clause.

The tables, views, and *table-reference*s [36] that might contain the column are called the *object tables* of the context. Two or more object tables might contain columns with the same name. One reason for qualifying a column name is to designate the object from which the column comes. For information about avoiding ambiguity between SQL parameters and variables and column names, see "References to SQL parameters and SQL variables" on page 1429.

A nested table expression which is preceded by a LATERAL or TABLE keyword will consider *table-reference*s that precede it in the FROM clause as object tables. The *table-reference*s that follow the nested table expression are not considered as object tables.

Table designators

A qualifier that designates a specific object table is called a *table designator*. The clause that identifies the object tables also establishes the table designators for them.

For example, the object tables of an expression in a SELECT clause are named in the FROM clause that follows it:

```
SELECT CORZ.COLA, OWNY.MYTABLE.COLA
  FROM OWNX.MYTABLE CORZ, OWNY.MYTABLE
```

Table designators in the FROM clause are established as follows:

- A name that follows a table or view name is both a correlation name and a table designator. Thus, CORZ is a table designator. CORZ is used to qualify the first column name in the select list.
- In SQL naming, an exposed table or view name is a table designator. Thus, OWNY.MYTABLE is a table designator. OWNY.MYTABLE is used to qualify the second column name in the select list.
- In system naming, the table designator for an exposed table or view name is the unqualified table or view name. In the following example MYTABLE is the table designator for OWNY/MYTABLE.

  ```
  SELECT CORZ.COLA, MYTABLE.COLA
    FROM OWNX/MYTABLE CORZ, OWNY/MYTABLE
  ```

Two or more object tables can be instances of the same table. In this case, distinct correlation names must be used to unambiguously designate the particular instances of the table. In the following FROM clause, X and Y are defined to refer, respectively, to the first and second instances of the table EMPLOYEE:

```
SELECT * FROM EMPLOYEE X,EMPLOYEE Y
```

Avoiding undefined or ambiguous references

When a column name refers to values of a column, it must be possible to resolve that column name to exactly one object table.

The following situations are considered errors:

- No object table contains a column with the specified name. The reference is undefined.
- The column name is qualified by a table designator, but the table designated does not include a column with the specified name. Again the reference is undefined.

36. In the case of a *joined-table*, each *table-reference* within the *joined-table* is an object table.

- The name is unqualified and more than one object table includes a column with that name. The reference is ambiguous.
- The column name is qualified by a table designator, but the table designated is not unique in the FROM clause and both occurrences of the designated table include the column. The reference is ambiguous.
- The column name is in a nested table expression which is not preceded by the LATERAL or TABLE keyword or a table function or nested table expression that is the right operand of a right outer join, full outer join, or a right exception join and the column name does not refer to a column of a *table-reference* within the nested table expression's fullselect. The reference is undefined.

Avoid ambiguous references by qualifying a column name with a uniquely defined table designator. If the column is contained in several object tables with different names, the object table names can be used as designators. Ambiguous references can also be avoided without the use of the table designator by giving unique names to the columns of one of the object tables using the column name list following the correlation name.

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name may be used. However, the qualifier used and the table used must be the same after fully qualifying the table name or view name and the table designator.

1. If the default schema is CORPDATA, then:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT
  FROM EMPLOYEE
```

   is a valid statement.

2. If the default schema is REGION, then:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT
  FROM EMPLOYEE                              ***INCORRECT***
```

   is invalid, because EMPLOYEE represents the table REGION.EMPLOYEE, but the qualifier for WORKDEPT represents a different table, CORPDATA.EMPLOYEE.

3. If the default schema is REGION, then:

```
SELECT EMPLOYEE.WORKDEPT
  FROM CORPDATA.EMPLOYEE                                ***INCORRECT***
```

   is invalid, because EMPLOYEE in the select list represents the table REGION.EMPLOYEE, but the explicitly qualified table name in the FROM clause represents a different table, CORPDATA.EMPLOYEE. In this case, either omit the table qualifier in the select list, or define a correlation name for the table designator in the FROM clause and use that correlation name as the qualifier for column names in the statement.

Column name qualifiers in correlated references

A *subselect* is a form of a query that can be used as a component of various SQL statements.

Refer to Chapter 5, "Queries," on page 627 for more information about subselects. A *subquery* is a form of a fullselect that is enclosed within parentheses. For example, a *subquery* can be used in a search condition. A fullselect used in the FROM clause of a query is called a *nested table expression*.

A subquery can include search conditions of its own, and these search conditions can, in turn, include subqueries. Therefore, an SQL statement can contain a hierarchy of subqueries. Those elements of the hierarchy that contain subqueries are said to be at a higher level than the subqueries they contain.

Every element of the hierarchy has a clause that establishes one or more table designators. This is the FROM clause, except in the highest level of an UPDATE or DELETE statement. A search condition, the select list, the join clause, an argument of a table function in a subquery, or a *nested table expression* that is preceded by the LATERAL keyword can reference not only columns of the tables identified by the FROM clause of its own element of the hierarchy, but also columns of tables identified at any level along the path from its own element to the highest level of the hierarchy. A reference to a column of a table identified at a higher level is called a *correlated reference*. A reference to a column of a table identified at the same level from a *nested table expression* through the use of the LATERAL keyword is called *lateral correlation*.

A correlated reference to column C of table T can be of the form C, T.C, or Q.C, if Q is a correlation name defined for T. However, a correlated reference in the form of an unqualified column name is not good practice. The following explanation is based on the assumption that a correlated reference is always in the form of a qualified column name and that the qualifier is a correlation name.

Q.C is a correlated reference only if these three conditions are met:

- Q.C is used in a search condition, select list, join clause, or an argument of a table function in a subquery.
- Q does not designate a table used in the FROM clause of that subquery, select list, join clause, or an argument of a table function in a subquery.
- Q does designate a table used at some higher level.

Q.C refers to column C of the table or view at the level where Q is used as the table designator of that table or view. Because the same table or view can be identified at many levels, unique correlation names are recommended as table designators. If Q is used to designate a table at more than one level, Q.C refers to the lowest level that contains the subquery that includes Q.C.

In the following statement, Q is used as a correlation name for T1 and T2, but Q.C refers to the correlation name associated with T2, because it is the lowest level that contains the subquery that includes Q.C.

```
SELECT *
  FROM T1 Q
  WHERE A < ALL (SELECT B
                   FROM T2 Q
                   WHERE B < ANY (SELECT D
                                    FROM T3
                                    WHERE D = Q.C))
```

Unqualified column names in correlated references

An unqualified column name can also be a correlated reference.

If the column:

- Is used in a search condition of a subquery
- Is not contained in a table used in the FROM clause of that subquery
- Is contained in a table used at some higher level

Unqualified correlated references are not recommended because it makes the SQL statement difficult to understand. The column will be implicitly qualified when the statement is prepared depending on which table the column was found in. Once this implicit qualification is determined it will not change until the statement is re-prepared. When an SQL statement that has an unqualified correlated reference is prepared or executed, a warning is returned (SQLSTATE 01545).

Variables

A *variable* in an SQL statement specifies a value that can be changed when the SQL statement is executed.

There are several types of *variable*s used in SQL statements:

global variable
Global variables are defined using the CREATE VARIABLE statement. For more information about how to refer to global variables see "Global variables."

host variable
Host variables are defined by statements of a host language. For more information about how to refer to host variables see "References to host variables" on page 149.

transition variable
Transition variables are defined in a trigger and refer to either the old or new values of columns. For more information about how to refer to transition variables see "CREATE TRIGGER" on page 1033.

SQL variable
SQL variables are defined by an SQL compound statement in an SQL function, SQL procedure, or trigger. For more information about SQL variables, see "References to SQL parameters and SQL variables" on page 1429.

SQL parameter
SQL parameters are defined in an CREATE FUNCTION (SQL Scalar), CREATE FUNCTION (SQL Table), or CREATE PROCEDURE (SQL) statement. For more information about SQL parameters, see "References to SQL parameters and SQL variables" on page 1429.

parameter marker
Variables cannot be referenced in dynamic SQL statements. Parameter markers are defined in an SQL descriptor and used instead. For more information about parameter markers, see Parameter Markers.

Global variables

Global variables are named memory variables that you can access and modify through SQL statements.

Global variables enable you to share relational data between SQL statements without the need for application logic to support this data transfer. You can control access to global variables through the GRANT (Global Variable Privileges) and REVOKE (Global Variable Privileges) statements.

DB2 supports created session global variables. A *session global variable* is associated with a specific session and contains a value that is unique to that session. A created session global variable is available to any active SQL statement running against the database on which the variable was defined. A session global variable can be associated with more than one session, but its value will be specific to each session. Created session global variables are defined in the system catalog.

Global variable names are qualified names. When a global variable is referenced without the schema name, the SQL path is used for name resolution.

For static SQL statements and SQL routines, global variables are resolved for a statement the first time all table references are resolved. In views, triggers, and other global variables, they are resolved when the object is created. If resolution were to be performed again on any global variable, it could change the behavior if, for example, a new global variable had been added with the same name in a different schema that is also in the SQL path.

Global variables that are referenced in dynamic statements will be resolved when the statement is initially prepared. They will not be resolved again unless the statement needs to be refreshed because a table has changed.

The name of a global variable can be the same as the name of a column in a table or view that is referenced in an SQL statement, as well as the name of an SQL variable or an SQL parameter in an SQL routine. Names that are the same should be explicitly qualified. If the name is not qualified, or it is qualified but is still ambiguous, the following rules describe the precedence of resolution:

- The name is checked to see if it is the name of a column of any existing table or view referenced in the statement at the current server.
- If used in an SQL routine, the name is checked to see if it is the name of an SQL variable, SQL parameter, or transition variable.
- If not found by either of these rules, it is assumed to be a global variable.
- If the SQL_GVAR_BUILD_RULE QAQQINI option is *EXIST and the global variable does not exist at precompile time or when an SQL routine is being created, an error will be issued.

When a global variable is referenced in a trigger, view, routine, or global variable, a dependency on the fully qualified global variable name is recorded for the statement or object. Also, if applicable, the authorization ID being used for the statement is checked for the appropriate privilege on the global variable.

Global variables can be used in any SQL statement that allows a variable. Global variables can be referenced within any expression except in the following situations:

- Check constraints
- Materialized query tables (MQTs)
- Derived index expressions
- A global variable is not allowed if the query specifies:
  - a distributed table,
  - a table with a read trigger, or
  - a logical file built over multiple physical file members.

Authorization: If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable if the global variable is referenced, and
  - The WRITE privilege on the global variable if the global variable is assigned a value, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

The value of a global variable can be changed using the SET, SELECT INTO, or VALUES INTO statement. It can also be changed if it is an argument of an OUT or INOUT parameter in a CALL statement.

References to host variables

A *host variable* is a COBOL data item, an RPG field, or a PLI, REXX, C++, C, or Java variable that is referenced in an SQL statement. Host variables are defined by statements of the host language.

Host variables cannot be referenced in dynamic SQL statements; instead, parameter markers must be used. For more information on parameter markers, see "Variables in dynamic SQL" on page 152.

A *host variable* in an SQL statement must identify a host variable described in the program according to the rules for declaring host variables.

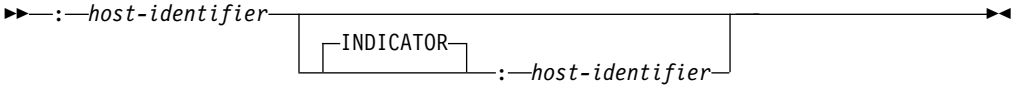
All host variables used in an SQL statement should be declared in an SQL declare section in all host languages other than Java, REXX, and RPG. Variables do not have to be declared in REXX. In Java and RPG, there is no declare section, and host variables may be declared throughout the program. No variables may be declared outside an SQL declare section with names identical to variables declared inside an SQL declare section. An SQL declare section begins with BEGIN DECLARE SECTION and ends with END DECLARE SECTION.

For further information about using host variables, see the Embedded SQL programming topic.

A variable in the INTO clause of a FETCH, a SELECT INTO, a SET variable, a GET DESCRIPTOR, or a VALUES INTO statement identifies a host variable to which a value from a result column is assigned. A variable in the GET DIAGNOSTICS statement identifies a host variable to which a diagnostic value is assigned. A host variable in a CALL or in an EXECUTE statement can be an output argument that is assigned a value after execution of the procedure, an input argument that provides an input value for the procedure, or both an input and output argument. In all other contexts a variable specifies a value to be passed to the database manager from the application program.

Non-Java variable references:

The general form of a *variable* reference in all languages other than Java is:



Each *host-identifier* must be declared in the source program. The variable designated by the second *host-identifier* is called an *indicator variable* and must have a data type of small integer. Indicator variables appear in two forms: *normal indicator variables* and *extended indicator variables*.

The purposes of a normal indicator variable are to:

- Specify a non-null value. A 0 (zero), or positive value of the indicator variable specifies that the associated, first, *host-identifier* provides the value of this host variable reference.

- Specify the null value. A negative value of the indicator variable specifies the null value.
- On output, indicate that one of the following numeric conversion errors:
  - Numeric conversion error (underflow or overflow)
  - Arithmetic expression error (division by 0)
  - A numeric value that is not valid

  A -2 value of the indicator variable indicates a null result because of one of these warnings.
- On output, indicate one of the following string errors:
  - Characters could not be converted
  - Mixed data not properly formed

  A -2 value of the indicator variable indicates a null result because of one of these warnings.
- On output, indicate one of the following datetime errors:
  - Date or timestamp conversion error (a date or timestamp that is not within the valid range of the dates for the specified format)
  - String representation of the datetime value is not valid

  A -2 value of the indicator variable indicates a null result because of one of these warnings.
- On output, indicate one of the following miscellaneous errors:
  - Argument of SUBSTR scalar function is out of range
  - Argument of a decryption function contains a data type that is not valid.

  A -2 value of the indicator variable indicates a null result because of one of these warnings.
- On output, record the original length of a string if the string is truncated on assignment to a host variable. If the string is truncated and there is no indicator variable, no error condition results.
- On output, record the seconds portion of a time if the time is truncated on assignment to a host variable. If the time is truncated and there is no indicator variable, no error condition results.

For example, if :V1:V2 is used to specify an insert or update value, and if V2 is negative, the value specified is the null value. If V2 is not negative the value specified is the value of V1.

Similarly, if :V1:V2 is specified in a CALL, FETCH, SELECT INTO, or VALUES INTO statement and the value returned is null, V1 is undefined, and V2 is set to a negative value. The negative value is:

- -1 if the value selected was the null value, or
- -2 if the null value was returned due to data mapping errors in the select list of an outer subselect. [37]

If the value returned is not null, that value is assigned to V1 and V2 is set to zero (unless the assignment to V1 requires string truncation, in which case, V2 is set to the original length of the string). If an assignment requires truncation of the seconds part of time, V2 is set to the number of seconds.

37. It should be noted that although the null value returned for data mapping errors can be returned on certain scalar functions and for arithmetic expressions, the result column is not considered null capable unless an argument of the arithmetic expression or scalar function is null capable.

If the second *host-identifier* is omitted, the *host variable* does not have an indicator variable. The value specified by the *host variable* :V1 is always the value of V1, and null values cannot be assigned to the variable. Thus, this form should not be used unless the corresponding result column cannot contain null values. If this form is used and the column contains nulls, the database manager will return an error at run-time (SQLSTATE 23502).

Extended indicator variables are limited to input host variables. Their purposes are:

- Specify a non-null value. A 0 (zero) or positive value specifies that the associated *host identifier* provides the value of this host variable reference.
- Specify the null value. A -1, -2, -3, -4, and -6 value specifies the null value.
- Specify the DEFAULT value. A -5 value specifies that the target column for this host variable is to be set to its default value.
- Specify an UNASSIGNED value. A -7 value specifies that the target column for this host variable is to be treated as if it hadn't been specified in the statement.

Extended indicator variables are only enabled if requested. Otherwise all indicator variables are normal indicator variables. In comparison to normal indicator variables, extended indicator variables have no additional restrictions for where null and non-null values can be used. Extended indicators can be used in indicator structures with host structures. Restrictions on where the extended indicator variable values of DEFAULT and UNASSIGNED are allowed apply uniformly, no matter how they are represented in the host application. The DEFAULT and UNASSIGNED extended indicator variables can only be used as an expression containing a single host parameter or a CAST of a single host parameter that is being assigned to a column. Output indicator variable values are never extended indicator variables.

When extended indicator variables are enabled, indicator variable values other than positive values, zero, and the six negative values listed previously must not be used. The DEFAULT and UNASSIGNED values must only appear in contexts where they are supported (INSERT, MERGE, and UPDATE statements).

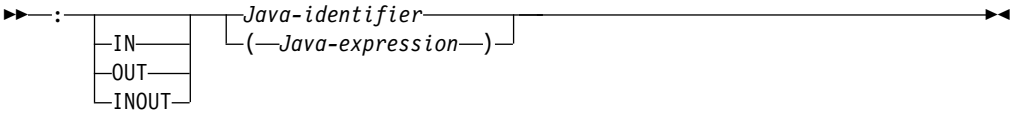
An SQL statement that references host variables in C, C++, ILE RPG, and PL/I, must be within the scope of the declaration of those host variables. For host variables referenced in the SELECT statement of a cursor, that rule applies to the OPEN statement rather than to the DECLARE CURSOR statement.

The CCSID of a string host variable is either:

- The CCSID specified in the DECLARE VARIABLE statement, or
- If a DECLARE VARIABLE with a CCSID clause is not specified for the host variable, the default CCSID of the application requester at the time the SQL statement that contains the host variable is executed unless the CCSID is for a foreign encoding scheme other than Unicode (such as ASCII). In this case, the host variable is converted to the default CCSID of the current server.

Java variable references:

The general form of a host variable reference in Java is:



In Java, indicator variables are not used. Instead, instances of a Java class can be set to a null value. Variables defined as Java primitive types cannot be set to a null value.

If IN, OUT, or INOUT is not specified, the default depends on the context in which the variable is used. If the Java variable is used in an INTO clause, OUT is the default. Otherwise, IN is the default. For more information about Java variables, see IBM Developer Kit for Java.

Example

Using the PROJECT table, set the host variable PNAME (VARCHAR(26)) to the project name (PROJNAME), the host variable STAFF (DECIMAL(5,2)) to the mean staffing level (PRSTAFF), and the host variable MAJPROJ (CHAR(6)) to the major project (MAJPROJ) for project (PROJNO) 'IF1000'. Columns PRSTAFF and MAJPROJ may contain null values, so provide indicator variables STAFF_IND (SMALLINT) and MAJPROJ_IND (SMALLINT).

```
SELECT PROJNAME, PRSTAFF, MAJPROJ
  INTO :PNAME, :STAFF :STAFF_IND, :MAJPROJ :MAJPROJ_IND
  FROM PROJECT
  WHERE PROJNO = 'IF1000'
```

Variables in dynamic SQL

In dynamic SQL statements, parameter markers are used instead of variables. A parameter marker is a question mark (?) that represents a position in a dynamic SQL statement where the application will provide a value; that is, where a variable would be found if the statement string were a static SQL statement.

The following examples shows a static SQL that uses host variables and a dynamic statement that uses parameter markers:

```
INSERT INTO DEPT
  VALUES( :HV_DEPTNO, :HV_DEPTNAME, :HV_MGRNO:IND_MGRNO, :HV_ADMRDEPT)

INSERT INTO DEPT
  VALUES( ?, ?, ?, ? )
```

For more information about parameter markers, see Parameter Markers, in "PREPARE" on page 1293.

References to LOB or XML variables

You can define regular LOB or XML variables, LOB or XML locator variables, and LOB or XML file reference variables.

Regular LOB or XML variables, LOB or XML locator variables, and LOB or XML file reference variables can be defined in the following host languages:

- C
- C++
- ILE RPG
- ILE COBOL
- PL/I (LOB only)

Where LOBs or XML are allowed, the term *variable* in a syntax diagram can refer to a regular variable, a locator variable, or a file reference variable. Since these variables are not native data types in host programming languages, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

When it is possible to define a variable that is large enough to hold an entire LOB or XML value and the performance benefit of delaying the transfer of data from the server is not required, a LOB or XML locator is not needed. However, it is often not acceptable to store an entire LOB or XML value in temporary storage due to host language restrictions, storage restrictions, or performance requirements. When storing a entire LOB or XML value at one time is not acceptable, a LOB or XML value can be referred to by a LOB or XML locator and portions of the LOB or XML value can be accessed.

References to LOB or XML locator variables

A LOB or XML *locator variable* is a variable that contains the locator representing a LOB or XML value on the application server.

LOB or XML locator variables can be defined in the following host languages:

- C
- C++
- ILE RPG
- ILE COBOL
- PL/I (LOB only)

See “Manipulating large objects with locators” on page 80 for information about how locators can be used to manipulate LOB and XML values.

A locator variable in an SQL statement must identify a LOB or XML locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement. For example, in C:

```
static volatile SQL TYPE IS CLOB_LOCATOR *loc1;
```

Like all other variables, a LOB or XML locator variable can have an associated indicator variable. Indicator variables for LOB or XML locator variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the variable is unchanged. When the indicator variable associated with a LOB or XML locator is null, the value of the referenced LOB or XML is null. This means that a locator can never point to a null value.

If a locator variable does not currently represent any value, an error occurs when the locator variable is referenced.

At transaction commit or any transaction termination, all LOB or XML locators that were acquired by the transaction are released.

It is the application programmer's responsibility to guarantee that any LOB or XML locator is only used in SQL statements that are executed at the same application server that originally generated the LOB or XML locator. For example, assume that a LOB locator is returned from one application server and assigned to a LOB locator variable. If that LOB locator variable is subsequently used in an SQL statement that is executed at a different application server, unpredictable results will occur.

References to LOB or XML file reference variables

A LOB or XML *file reference variable* is used for direct file input and output for a LOB or XML.

A LOB or XML *file reference variable* can be defined in the following host languages:

- C
- C++
- ILE RPG
- ILE COBOL
- PL/I (LOB only)

Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

A file reference variable represents (rather than contains) the file, just as a LOB or XML locator represents, rather than contains, the LOB or XML data. Database queries, updates, and inserts may use file reference variables to store or to retrieve single column values. The file referenced must exist at the application requester.

Like all other variables, a file reference variable can have an associated indicator variable. Indicator variables for file reference variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the variable is unchanged. When the indicator variable associated with a file reference variable is null, the value of the referenced LOB or XML is null. This means that a file reference variable can never point to a null value.

The length attribute of a file reference variable is assumed to be the maximum length of a LOB or XML.

File reference variables are currently supported in the root (/), QOpenSys, and UDFS file systems. When a file is created, it is given the CCSID of the data that is being written to the file. Currently, mixed CCSIDs are not supported. To use a file created with a file reference variable, the file should be opened in binary mode.

For more information about file reference variables, see the SQL programming topic collection.

References to XML variables

XML variables are defined in host languages as a type of LOB variable.

Like LOBs, an XML variable can be defined as a string, locator, or file reference variable. It can be treated as CLOB, DBCLOB, or BLOB data. XML-supported CCSIDs can be assigned for CLOB and DBCLOB variables. This value is used to

define the encoding of the XML data stored within the LOB variable. An XML variable defined as a BLOB will contain data that is encoded as specified within the data according to the XML 1.0 specification for determining encoding. For example, an XML variable that uses a CLOB storage structure can be defined in C as follows:

```
SQL TYPE IS XML AS CLOB(10K);
```

Although the application's XML variable declaration includes a LOB type specification, the variable declarations are considered the XML data type, not the LOB type that is used in the declaration. The application can also use non-XML variables in place of XML variables. For example, when a prepared statement is executed, the application might use a character variable to replace an XML parameter marker in the statement.

Although the XML data type is incompatible with all other data types, both XML and non-XML data types can be used for input to and output from XML data. Applications can use character, Unicode graphic, or binary variables as input or output variables in SQL statements.

Whenever an XML variable is used as input to an SQL statement, the value is implicitly parsed.

Host structures

A *host structure* is a COBOL group, PL/I, C, or C++ structure, or RPG data structure that is referenced in an SQL statement. In Java and REXX, there is no equivalent to a *host structure*.

Host structures are defined by statements of the host language, as explained in the Embedded SQL programming topic collection. As used here, the term host structure does not include an SQLCA or SQLDA.

The form of a host structure reference is identical to the form of a host variable reference. The reference :S1:S2 is a host structure reference if S1 names a host structure. If S1 designates a host structure, S2 must be either a small integer variable, or an array of small integer variables. S1 is the host structure and S2 is its indicator array.

A host structure can be referenced in any context where a list of host variables can be referenced. A host structure reference is equivalent to a reference to each of the host variables contained within the structure in the order which they are defined in the host language structure declaration. The *n*th variable of the indicator array is the indicator variable for the *n*th variable of the host structure.

In C, for example, if V1, V2, and V3 are declared as variables within the structure S1, the statement:

```
EXEC SQL FETCH CURSOR1 INTO :S1;
```

is equivalent to:

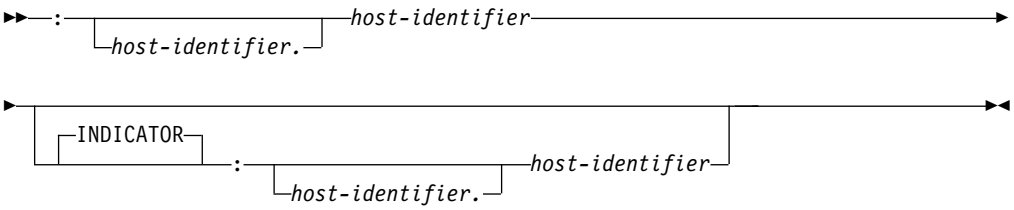
```
EXEC SQL FETCH CURSOR1 INTO :V1, :V2, :V3;
```

If the host structure has *m* more variables than the indicator array, the last *m* variables of the host structure do not have indicator variables. If the host structure has *m* fewer variables than the indicator array, the last *m* variables of the indicator array are ignored. These rules also apply if a reference to a host structure includes

an indicator variable or if a reference to a host variable includes an indicator array. If an indicator array or indicator variable is not specified, no variable of the host structure has an indicator variable.

In addition to structure references, individual host variables in the host structure or indicator variables in the indicator array can be referenced by qualified names. The qualified form is a host identifier followed by a period and another host identifier. The first host identifier must name a host structure, and the second host identifier must name a host variable within that host structure.

The general form of a host variable or host structure reference is:



A *host-variable* in an expression must identify a host variable (not a structure) described in the program according to the rules for declaring host variables.

The following C example shows a references to host structure, host indicator array, and a host variable:

```
struct { char empno[7];
                struct          { short int firstname_len;
                                  char firstname_text[12];
                                } firstname;
                char midint,
                struct          { short int lastname_len;
                                  char lastname_text[15];
                                } lastname;
                char workdept[4];
       } pemp1;
short ind[14];
short eind
struct { short  ind1;
         short  ind2;
       } indstr;

  .....
strcpy(pemp1.empno,"000220");
.....
EXEC SQL
  SELECT *
    INTO :pemp1:ind
    FROM corpdata.employee
    WHERE empno=:pemp1.empno;
```

In the example above, the following references to host variables and host structures are valid:

```
:pemp1   :pemp1.empno   :pemp1.empno:eind   :pemp1.empno:indstr.ind1
```

For more information about how to refer to host structures in C, C++, COBOL, PL/I, and RPG, see the Embedded SQL programming topic collection.

Host structure arrays

In PL/I, C++, and C, a host structure array is a structure name having a dimension attribute. In COBOL, it is a one-dimensional table. In RPG, it is an occurrence data structure. In ILE RPG, it can also be a data structure with the keyword DIM.

A host structure array can only be referenced in the FETCH statement when using a multiple-row fetch, or in an INSERT statement when using a multiple-row insert. Host structure arrays are defined by statements of the host language, as explained in the Embedded SQL programming topic collection.

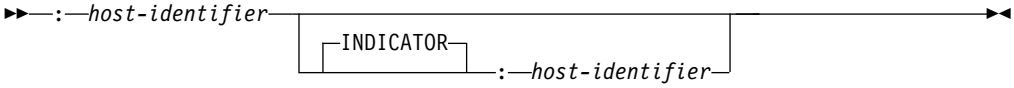
The form of a host structure array is identical to the form of a host variable reference. The reference :S1:S2 is a reference to host structure array if S1 names a host structure array. If S1 designates a host structure, S2 must be either a small integer host variable, an array of small integer host variables, or a two dimensional array of small integer host variables. In the following example, S1 is the host structure array and S2 is its indicator array.

```
EXEC  SQL FETCH CURSOR1 FOR 5 ROWS
          INTO :S1:S2;
```

The dimension of the host structure and the indicator array must be equal.

If the host structure has *m* more variables than the indicator array, the last *m* variables of the host structure do not have indicator variables. If the host structure has *m* fewer variables than the indicator array, the last *m* variables of the indicator array are ignored. If an indicator array or variable is not specified, no variable of the host structure array has an indicator variable.

The following diagram specifies the syntax of references to an array of host structures:



Arrays of host structures are not supported in REXX.

Functions

A *function* is an operation denoted by a function name followed by one or more operands that are enclosed in parentheses. It represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*. For example, a function can be passed two input arguments that have date and time data types and return a value with a timestamp data type as the result.

Types of functions

There are several ways to classify functions.

One way to classify functions is as built-in, user-defined, or generated user-defined functions for distinct types.

- *Built-in functions* are functions that come with the database manager. These functions provide a single-value result. Built-in functions include operator functions such as "+", aggregate functions such as AVG, and scalar functions such as SUBSTR. For a list of the built-in aggregate and scalar functions and information about these functions, see Chapter 3, “Built-in functions,” on page 227.

  The *built-in functions* are part of schema QSYS2. [38]
- *User-defined functions* are functions that are created using the CREATE FUNCTION statement and registered to the database manager in catalog table QSYS2.SYSROUTINES and catalog view QSYS2.SYSFUNCS. For more information, see “CREATE FUNCTION” on page 851. These functions allow users to extend the function of the database manager by adding their own or third party vendor function definitions.

  A user-defined function is either an *SQL*, *external*, or *sourced* function. An SQL function is defined to the database using only SQL statements. An external function is defined to the database with a reference to an external program or service program that is executed when the function is invoked. A sourced function is defined to the database with a reference to a built-in function or another user-defined function. Sourced functions can be used to extend built-in aggregate and scalar functions for use on distinct types.

  A user-defined function resides in the schema in which it was created. The schema cannot be QSYS, QSYS2, or QTEMP.
- *Generated user-defined functions for distinct types* are functions that the database manager automatically generates when a distinct type is created using the CREATE TYPE statement. These functions support casting from the distinct type to the source type and from the source type to the distinct type. The ability to cast between the data types is important because a distinct type is compatible only with itself.

  The generated cast functions reside in the same schema as the distinct type for which they were created. The schema cannot be QSYS, QSYS2, or QTEMP. For more information about the functions that are generated for a distinct type, see “CREATE TYPE (Distinct)” on page 1055.

Another way to classify functions is as aggregate, scalar, or table functions, depending on the input data values and result values.

38. *Built-in functions* are implemented internally by the database manager, so an associated program or service program object does not exist for a *built-in function*. Furthermore, the catalog does not contain information about *built-in functions*. However, *built-in functions* can be treated as if they exist in QSYS2 and a *built-in function* name can be qualified with QSYS2.

- An *aggregate function* receives a set of values for each argument (such as the values of a column) and returns a single-value result for the set of input values. Aggregate functions are sometimes called *column functions*. Built-in functions and user-defined sourced functions can be aggregate functions.
- A *scalar function* receives a single value for each argument and returns a single-value result. Built-in functions and user-defined functions can be scalar functions. Generated user-defined functions for distinct types are also scalar functions.
- A *table function* returns a table for the set of arguments it receives. Each argument is a single value. A table function can only be referenced in the FROM clause of a subselect. A table function can be defined as an external function or as an SQL function, but a table function cannot be a sourced function.

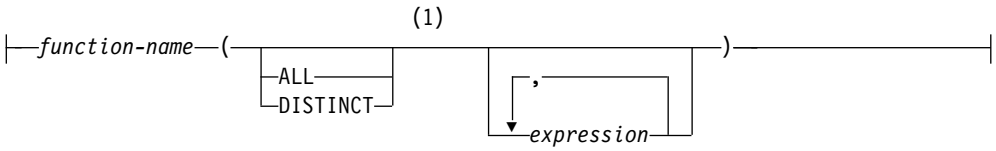
  Table functions can be used to apply SQL language processing power to data that is not stored in the database or to allow access to such data as if it were stored in a table. For example, a table function can read a file, get data from the Web, or access a Lotus Notes® database and return a result table.

Function invocation

Each reference to a scalar or aggregate function (either built-in or user-defined) conforms to specific syntax.

The syntax is as follows: [39]

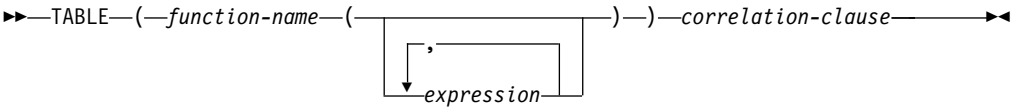
function-invocation:



Notes:

1 The ALL or DISTINCT keyword can be specified only for an aggregate function or a user-defined function that is sourced on an aggregate function.

Each reference to a table function conforms to the following syntax:



In the above syntax, *expression* is the same as it is for a scalar or aggregate function. See “Expressions” on page 165 for other rules for *expression*.

When the function is invoked, the value of each of its parameters is assigned, using storage assignment, to the corresponding parameter of the function. Control is passed to external functions according to the calling conventions of the host language. When execution of a user-defined aggregate or scalar function is

39. A few functions allow keywords instead of expressions. For example, the CHAR function allows a list of keywords to indicate the desired date format. A few functions use keywords instead of commas in a comma separated list of expressions. For example, the EXTRACT, TRIM, and POSITION functions use keywords.

complete, the result of the function is assigned, using storage assignment, to the result data type. For details on the assignment rules, see "Assignments and comparisons" on page 98.

Table functions can be referenced only in the FROM clause of a subselect. For more details on referencing a table function, see the description of the FROM clause in "table-reference" on page 633.

Function resolution

A function is invoked by its function name, which is implicitly or explicitly qualified with a schema name, followed by parentheses that enclose the arguments to the function.

Within the database, each function is uniquely identified by its function signature, which is its schema name, function name, the number of parameters, and the data types of the parameters. Thus, a schema can contain several functions that have the same name but each of which have a different number of parameters, or parameters with different data types. Or, a function with the same name, number of parameters, and types of parameters can exist in multiple schemas. When any function is invoked, the database manager must determine which function to execute. This process is called *function resolution*.

Function resolution is similar for functions that are invoked with a qualified or unqualified function name with the exception that for an unqualified name, the database manager needs to search more than one schema.

- *Qualified function resolution:* When a function is invoked with a function name and a schema name, the database manager only searches the specified schema to resolve which function to execute. The database manager selects candidate functions based on the following criteria:
  - The name of the function instance matches the name in the function invocation.
  - The number of input parameters in the function instance matches the number of arguments in the function invocation.
  - The authorization ID of the statement must have the EXECUTE privilege to the function instance.
  - The data type of each input argument of the function invocation matches or is *promotable* to the data type of the corresponding parameter of the function instance.

  If no function in the schema meets these criteria, an error is returned. If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns an integer data type where a character data type is required, or returns a table where a table is not allowed, an error is returned.
- *Unqualified function resolution:* When a function is invoked with only a function name, the database manager needs to search more than one schema to resolve the function instance to execute. The SQL path contains the list of schemas to search. For each schema in the SQL path (see "SQL path" on page 63), the database manager selects candidate functions based on the following criteria:
  - The name of the function instance matches the name in the function invocation.
  - The number of input parameters in the function instance matches the number of function arguments in the function invocation.

- The authorization ID of the statement must have the EXECUTE privilege to the function instance.
- The data type of each input argument of the function invocation matches or is *promotable* to the data type of the corresponding parameter of the function instance.

If no function in the schema meets these criteria, an error is returned. If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns an integer data type where a character data type is required, or returns a table where a table is not allowed, an error is returned.

After the database manager identifies the candidate functions, it selects the candidate with the best fit as the function instance to execute (see "Determining the best fit"). If more than one schema contains the function instance with the best fit (the function signatures are identical except for the schema name), the database manager selects the function whose schema is earliest in the SQL path.

Function resolution applies to all functions, including built-in functions. Built-in functions logically exist in schema QSYS2. If schema QSYS2 is not explicitly specified in the SQL path, the schema is implicitly assumed at the front of the path. When an unqualified function name is specified, the SQL path must be set to a list of schemas in the desired search order so that the intended function is selected.

In a CREATE VIEW or CREATE TABLE statement, function resolution occurs at the time the view or materialized table is created. If another function with the same name is subsequently created, the view or materialized query table is not affected, even if the new function is a better fit than the one chosen at the time the view or materialized table was created. In a CREATE FUNCTION, CREATE PROCEDURE, or CREATE TRIGGER statement, function resolution occurs at the time the function, procedure, or trigger is created. The schema of the function that was chosen is saved in the function, procedure, or trigger. If another function with the same name is subsequently created, the function, procedure, or trigger is only affected if the new function is created in the same schema and is a better fit than the one chosen at the time the function, procedure, or trigger was created.

Determining the best fit

There might be more than one function with the same name that is a candidate for execution. In that case, the database manager determines which function is the best fit for the invocation by comparing the argument and parameter data types. Note that the data type of the result of the function or the type of function (aggregate, scalar, or table) under consideration does not enter into this determination.

If the data types of all the parameters for a given function are the same as those of the arguments in the function invocation, that function is the best fit. When determining whether the data types of the parameters are the same as the arguments:

- Synonyms of data types match. For example, DOUBLE and FLOAT are considered to be the same.
- Attributes of a data type such as length, precision, scale, and CCSID are ignored. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), and DECIMAL(4,3).

- The character and graphic types are considered to be the same. For example, the following are considered to be the same type: CHAR and GRAPHIC, VARCHAR and VARGRAPHIC, and CLOB and DBCLOB. CHAR(13) and GRAPHIC(8) are considered to be the same type.

If there is no match, the database manager compares the data types in the parameter lists from left to right, using the following method:

1. Compare the data type of the first argument in the function invocation to the data type of the first parameter in each function. (The rules previously mentioned are used to determine whether the data types are the same.)
2. For this argument, if one function has a data type that fits the function invocation better than the data types in the other functions, that function is the best fit. The precedence list for the promotion of data types in "Promotion of data types" on page 93 shows the data types that fit each data type in best-to-worst order.
3. If the data type of the first parameter for more than one candidate function fits the function invocation equally well, repeat this process for the next argument of the function invocation. Continue for each argument until a best fit is found.

The following examples illustrate function resolution.

Example 1: Assume that MYSCHEMA contains two functions, both named FUNA, that were created with these partial CREATE FUNCTION statements.

```
CREATE FUNCTION MYSCHEMA.FUNA (VARCHAR(10), INT, DOUBLE) ...
CREATE FUNCTION MYSCHEMA.FUNA (VARCHAR(10), REAL, DOUBLE) ...
```

Also assume that a function with three arguments of data types VARCHAR(10), SMALLINT, and DECIMAL is invoked with a qualified name:

```
MYSCHEMA.FUNA( VARCHARCOL, SMALLINTCOL, DECIMALCOL ) ...
```

Both MYSCHEMA.FUNA functions are candidates for this function invocation because they meet the criteria specified in "Function resolution" on page 160. The data types of the first parameter for the two function instances in the schema, which are both VARCHAR, fit the data type of the first argument of the function invocation, which is VARCHAR, equally well. However, for the second parameter, the data type of the first function (INT) fits the data type of the second argument (SMALLINT) better than the data type of second function (REAL). Therefore, the database manager selects the first MYSCHEMA.FUNA function as the function instance to execute.

Example 2: Assume that functions were created with these partial CREATE FUNCTION statements:

```
1. CREATE FUNCTION SMITH.ADDIT (CHAR(5), INT, DOUBLE) ...
2. CREATE FUNCTION SMITH.ADDIT (INT, INT, DOUBLE) ...
3. CREATE FUNCTION SMITH.ADDIT (INT, INT, DOUBLE, INT) ...
4. CREATE FUNCTION JOHNSON.ADDIT (INT, DOUBLE, DOUBLE) ...
5. CREATE FUNCTION JOHNSON.ADDIT (INT, INT, DOUBLE) ...
6. CREATE FUNCTION TODD.ADDIT (REAL) ...
7. CREATE FUNCTION TAYLOR.SUBIT (INT, INT, DECIMAL) ...
```

Also assume that the SQL path at the time an application invokes a function is "TAYLOR", "JOHNSON", "SMITH". The function is invoked with three data types (INT, INT, DECIMAL) as follows:

```
SELECT  ... ADDIT(INTCOL1, INTCOL2, DECIMALCOL) ...
```

Function 5 is chosen as the function instance to execute based on the following evaluation:

- Function 6 is eliminated as a candidate because schema TODD is not in the SQL path.
- Function 7 in schema TAYLOR is eliminated as a candidate because it does not have the correct function name.
- Function 1 in schema SMITH is eliminated as a candidate because the INT data type is not promotable to the CHAR data type of the first parameter of Function 1.
- Function 3 in schema SMITH is eliminated as a candidate because it has the wrong number of parameters.
- Function 2 is a candidate because the data types of its parameters match or are promotable to the data types of the arguments.
- Both Function 4 and 5 in schema JOHNSON are candidates because the data types of their parameters match or are promotable to the data types of the arguments. However, Function 5 is chosen as the better candidate because although the data types of the first parameter of both functions (INT) match the first argument (INT), the data type of the second parameter of Function 5 (INT) is a better match of the second argument (INT) than the data type of Function 4 (DOUBLE).
- Of the remaining candidates, Function 2 and 5, the database manager selects Function 5 because schema JOHNSON comes before schema SMITH in the SQL path.

Example 3: Assume that functions were created with these partial CREATE FUNCTION statements:

```
1. CREATE FUNCTION BESTGEN.MYFUNC (INT, DECIMAL(9,0)) ...
2. CREATE FUNCTION KNAPP.MYFUNC (INT, NUMERIC(8,0))...
3. CREATE FUNCTION ROMANO.MYFUNC (INT, NUMERIC(8,0))...
4. CREATE FUNCTION ROMANO.MYFUNC (INT, FLOAT) ...
```

Also assume that the SQL path at the time an application invokes a function is "ROMANO", "KNAPP", "BESTGEN" and that the authorization ID of the statement has the EXECUTE privilege to Functions 1, 2, and 4. The function is invoked with two data types (SMALLINT, DECIMAL) as follows:

```
SELECT  ... MYFUNC(SINTCOL1, DECIMALCOL) ...
```

Function 2 is chosen as the function instance to execute based on the following evaluation:

- Function 3 is eliminated. It is not a candidate for this function invocation because the authorization ID of the statement does not have the EXECUTE privilege to the function. The remaining three functions are candidates for this function invocation because they meet the criteria specified in “Function resolution” on page 160.
- Function 4 in schema ROMANO is eliminated because the second parameter (FLOAT) is not as good a fit for the second argument (DECIMAL) as the second parameter of either Function 1 (DECIMAL) or Function 2 (NUMERIC).
- The second parameters of Function 1 (DECIMAL) and Function 2 (NUMERIC) are equally good fits for the second argument (DECIMAL).
- Function 2 is finally chosen because "KNAPP" precedes "BESTGEN" in the SQL path.

Best fit considerations

Once the function is selected, there are still possible reasons why the use of the function may not be permitted. Each function is defined to return a result with a specific data type. If this result data type is not compatible within the context in which the function is invoked, an error will occur.

For example, given functions named STEP defined with different data types as the result:

```
STEP(SMALLINT) RETURNS CHAR(5)
STEP(DOUBLE) RETURNS INTEGER
```

and the following function reference (where S is a SMALLINT column):

```
SELECT ... 3 +STEP(S)
```

then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement because the result type is CHAR(5) instead of a numeric type as required for an argument of the addition operator.

In cases where the arguments of the function invocation were not an exact match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns (see "Assignments and comparisons" on page 98). This includes the case where precision, scale, length, or CCSID differs between the argument and the parameter.

An error also occurs in the following examples:

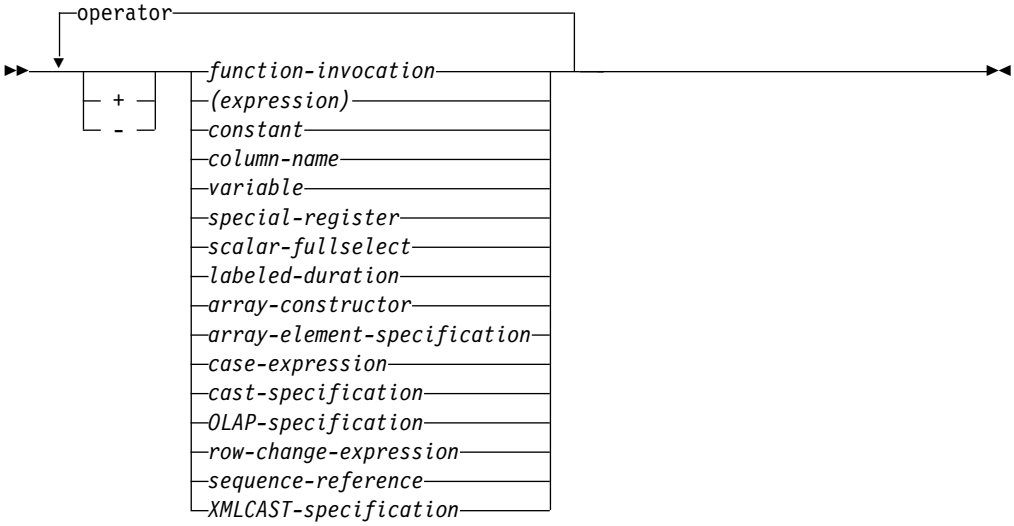
- The function is referenced in the TABLE clause of a FROM clause, but the function selected by the function resolution step is a scalar or aggregate function.
- The function referenced in an SQL statement requires a scalar or aggregate function, but the function selected by the function resolution step is a table function.

Expressions

An expression specifies a value.

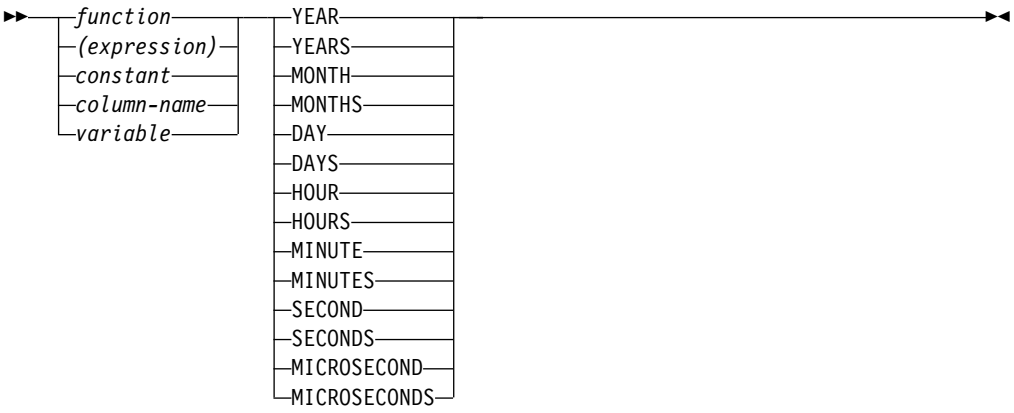
Authorization: The use of some of the expressions, such as a *scalar-fullselect*, *sequence-reference*, *function-invocation*, or *cast-specification* may require having the appropriate authorization. For these expressions, the privileges held by the authorization ID of the statement must include the following authorization:

- *global variable*. For information about authorization considerations, see "Global variables" on page 147.
- *scalar-fullselect*. For information about authorization considerations, see Chapter 5, "Queries," on page 627.
- *sequence-reference*. The authorization to reference the sequence. For information about authorization considerations, see "Notes" on page 196.
- *function-invocation*. The authorization to execute a user-defined function. For information about authorization considerations, see "Function invocation" on page 159.
- *array-element-specification*. The authorization to reference an array type. For information about authorization considerations, see "GRANT (Type Privileges)" on page 1242.
- *cast-specification*. The authorization to reference a user-defined type. For information about authorization considerations, see "CAST specification" on page 185.



operator:

labeled-duration:



Without operators

If no operators are used, the result of the expression is the specified value.

Example

```
SALARY     :SALARY     'SALARY'    MAX(SALARY)
```

With arithmetic operators

If arithmetic operators are used, the result of the expression is a number derived from the application of the operators to the values of the operands.

If any operand can be null, the result can be null. If any operand has the null value, the result of the expression is the null value.

If one operand of an arithmetic operator is numeric, the other operand can be a string. The string is first converted to the data type of the numeric operand and must contain a valid string representation of a number.

The prefix operator + (*unary plus*) does not change its operand. The prefix operator - (*unary minus*) reverses the sign of a nonzero non-decimal floating-point operand. The prefix operator - (*unary minus*) reverses the sign of all decimal floating-point operands, including zero and special values; that is, signalling and non-signalling NaNs and plus and minus infinity. If the data type of A is small integer, the data type of - A is large integer. The first character of the token following a prefix operator must not be a plus or minus sign.

The *infix operators*, +, -, *, /, and **, specify addition, subtraction, multiplication, division, and exponentiation, respectively. The value of the second operand of division must not be zero, except if the calculation is performed using decimal floating-point arithmetic and the first operand is infinity or -infinity.

In COBOL, blanks must precede and follow a minus sign to avoid any ambiguity with COBOL host variable names (which allow use of a dash).

The result of an exponentiation (**) operator is a double-precision floating-point number. The result of the other operators depends on the type of the operand.

Operands with a NUMERIC data type are converted to DECIMAL operands prior to performing the arithmetic operation.

Two integer operands

If both operands of an arithmetic operator are integers with zero scale, the operation is performed in binary, and the result is a large integer unless either (or both) operand is a big integer, in which case the result is a big integer. Any remainder of division is lost. The result of an integer arithmetic operation (including *unary minus*) must be within the range of large or big integers. If either integer operand has nonzero scale, it is converted to a decimal operand with the same precision and scale.

Integer and decimal operands

If one operand is an integer with zero scale and the other is decimal, the operation is performed in decimal using a temporary copy of the integer that has been converted to a decimal number with precision and scale 0.

This is defined in the following table:

Operand	Precision of Decimal Copy
Column or variable: big integer	19
Column or variable: large integer	11
Column or variable: small integer	5
Constant (including leading zeros)	Same as the number of digits in the constant

If one operand is an integer with nonzero scale, it is first converted to a decimal operand with the same precision and scale.

Two decimal operands

If both operands are decimal, the operation is performed in decimal. The result of any decimal arithmetic operation is a decimal number with a precision and scale that are dependent on the operation and the precision and scale of the operands. If the operation is addition or subtraction and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands. The copy of the shorter operand is extended with trailing zeros so that its fractional part has the same number of digits as the longer operand.

Unless specified otherwise, all functions and operations that accept decimal numbers allow a precision of up to 63 digits. The result of a decimal operation must not have a precision greater than 63.

Decimal arithmetic in SQL

The following formulas define the precision and scale of the result of decimal operations in SQL. The symbols *p* and *s* denote the precision and scale of the first operand and the symbols *p'* and *s'* denote the precision and scale of the second operand.

The symbol *mp* denotes the maximum precision. The value of *mp* is 63 if:

- either *p* or *p'* is greater than 31, or
- a value of 63 was explicitly specified for the maximum precision.

Otherwise, the value of *mp* is 31.

The symbol *ms* denotes the maximum scale. The default value of *ms* is 31. *ms* can be explicitly set to any number from 0 to the maximum precision.

The symbol *mds* denotes the minimum divide scale. The default value of *mds* is 0, where 0 indicates that no minimum scale is specified. *mds* can be explicitly set to any number from 1 to min (*ms*, 9).

The maximum precision, maximum scale, and minimum divide scale can be explicitly specified on the DECRESULT parameter of the CRTSQLxxx command, RUNSQLSTM command, or SET OPTION statement. They can also be specified in ODBC data sources, JDBC properties, OLE DB properties, .NET properties.

Addition and subtraction

The scale of the result of addition and subtraction is max (*s*,*s'*). The precision is min(*mp*,max(*p-s*,*p'-s'*) +max(*s*,*s'*)+1).

Multiplication

The precision of the result of multiplication is min (*mp*,*p+p'*) and the scale is min(*ms*,*s+s'*).

Division

The precision of the result of division is (*p-s+s'*) + max(*mds*, min(*ms*, *mp* - (*p-s+s'*))). The scale is max(*mds*, min(*ms*, *mp* - (*p-s+s'*))). The scale must not be negative.

Floating-point operands

If either operand of an arithmetic operator is floating point and neither operand is decimal floating-point, the operation is performed in floating point. The operands are first converted to double-precision floating-point numbers, if necessary. Thus, if any element of an expression is a floating-point number, the result of the expression is a double-precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer converted to double-precision floating point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number that has been converted to double-precision floating point. The result of a floating-point operation must be within the range of floating-point numbers.

The order in which floating-point operands (or arguments to functions) are processed can slightly affect results because floating-point operands are approximate representations of real numbers. Since the order in which operands are processed may be implicitly modified by the optimizer (for example, the optimizer may decide what degree of parallelism to use and what access plan to use), an application should not depend on the results being precisely the same each time an SQL statement is executed that uses floating-point operands.

Decimal floating-point operands

If either operand of an arithmetic operator is decimal floating-point, the operation is performed in decimal floating-point.

Integer and DECFLOAT(n) operands

If one operand is a small integer or integer and the other is DECFLOAT, the operation is performed in DECFLOAT(n) using a temporary copy of the integer that has been converted to a DECFLOAT(n) number. If one operand is a big integer and the other is DECFLOAT, then a temporary copy of the big integer is converted

to a DECFLOAT(34) number. The rules for two DECFLOAT operands are then applied.

Decimal and DECFLOAT(n) operands

If one operand is a decimal and the other is DECFLOAT, the operation is performed in DECFLOAT using a temporary copy of the decimal number that has been converted to a DECFLOAT number based on the precision of the decimal number. If the decimal number has a precision < 17, the decimal number is converted to DECFLOAT(16). Otherwise, the decimal number is converted to a DECFLOAT(34) number. The rules for two DECFLOAT operands are then applied.

Floating-point and DECFLOAT(n) operands

If one operand is floating-point (real or double) and the other is DECFLOAT, the operation is performed in DECFLOAT(n) using a temporary copy of the floating-point number that has been converted to a DECFLOAT(n) number.

Two DECFLOAT operands

If both operands are DECFLOAT(n), the operation is performed in DECFLOAT(n). If one operand is DECFLOAT(16) and the other is DECFLOAT(34), the operation is performed in DECFLOAT(34).

General arithmetic operation rules for DECFLOAT

The following general rules apply to all arithmetic operations on the DECFLOAT data type.

- Every operation on finite numbers is carried out (as described under the individual operations) as though an exact mathematical result is computed, using integer arithmetic on the coefficient where possible.

  If the coefficient of the theoretical exact result has no more than the number of digits that reflect its precision (16 or 34), then (unless there is an underflow or overflow) it is used for the result without change. Otherwise, it is rounded (shortened) to exactly the number of digits that reflect its precision (16 or 34) and the exponent is increased by the number of digits removed.

  Rounding uses the DECFLOAT rounding mode. For more information, see "CURRENT DECFLOAT ROUNDING MODE" on page 133.

  If the value of the adjusted exponent of the result is less than E_{min}, then a subnormal warning is returned.[40] In this case, the calculated coefficient and exponent form the result, unless the value of the exponent is less than E_{tiny}, in which case the exponent will be set to E_{tiny}, the coefficient is rounded (possibly to zero) to match the adjustment of the exponent, and the sign is unchanged. If this rounding gives an inexact result then an underflow warning is returned. [40]

  If the value of the adjusted exponent of the result is larger than E_{max} then an overflow warning is returned. In this case, the result may be infinite. It will have the same sign as the theoretical result.
- Arithmetic using the special value Infinity follows the usual rules, where negative Infinity is less than every finite number and positive Infinity is greater than every finite number. Under these rules, an infinite result is always exact. The following arithmetic operations return a warning and result in NaN:[40]
  - Add +infinity to -infinity during an addition or subtraction operation
  - Multiply 0 or -0 by +infinity or -infinity

40. The warning is only returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

  - Divide either +infinity or -infinity by either +infinity or -infinity
- Signaling NaNs always raise a warning or error when used as an operand of an arithmetic operation and NaNs are returned.
- The result of an arithmetic operation which has an operand which is a NaN, is NaN. The sign of the result is copied from the first operand which is a NaN. Whenever the result is a NaN, the sign of the result depends only on the copied operand.
- The sign of the result of a multiplication or division will be negative only if the operands have different signs and neither is a NaN.
- The sign of the result of an addition or subtraction will be negative only if the result is less than zero and neither operand is a NaN.

In some instances, negative zero might be the result from arithmetic operations and numeric functions.

Examples involving special values

```
INFINITY + 1             = INFINITY
INFINITY + INFINITY      = INFINITY
INFINITY + -INFINITY     = NAN       -- warning
NAN + 1                  = NAN
NAN + INFINITY           = NAN
1 - INFINITY             = -INFINITY
INFINITY - INFINITY      = NAN       -- warning
-INFINITY - -INFINITY    = NAN       -- warning
-0.0 - 0.0E1             = -0.0
-1.0 * 0.0E1             = -0.0
1.0E1 / 0                = INFINITY
-1.0E5 / 0.0             = -INFINITY
1.0E5 / -0               = -INFINITY
INFINITY / -INFINITY     = NAN       -- warning
INFINITY / 0             = INFINITY  -- warning
-INFINITY / 0            = -INFINITY -- warning
-INFINITY / -0           = INFINITY  -- warning
```

Distinct type operands

A distinct type cannot be used with arithmetic operators even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, then the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE ( INCOME, EXPENSES )
  RETURNS DECIMAL(8,2) SOURCE "-" ( DECIMAL, DECIMAL)
```

Alternately, the - (minus) operator could be overloaded using a user-defined function to subtract the new data types.

```
CREATE FUNCTION "-" ( INCOME, EXPENSES )
  RETURNS DECIMAL(8,2) SOURCE "-" ( DECIMAL, DECIMAL)
```

Alternatively, the distinct type can be cast to a built-in type, and the result can be used as an operand of an arithmetic operator.

With the concatenation operator

If the concatenation operator (CONCAT or ||) is used, the result of the expression is a string.

The operands of concatenation must be compatible strings or numeric data types.[41] The operands must not be distinct types. If a numeric operand is specified, it is CAST to the equivalent character string prior to concatenation. Note that a binary string cannot be concatenated with a character string unless the character string is defined as FOR BIT DATA.

The data type of the result is determined by the data types of the operands. The data type of the result is summarized in the following table:

Table 26. Result Data Types With Concatenation

If one operand column is ...	And the other operand is ...	The data type of the result column is ...
DBCLOB(x)	CHAR(y)* or VARCHAR(y)* or CLOB(y)* or GRAPHIC(y) or VARGRAPHIC(y) or DBCLOB(y)	DBCLOB(z) where z = MIN(x + y, maximum length of a DBCLOB)
VARGRAPHIC(x)	CHAR(y)* or VARCHAR(y)* or GRAPHIC(y) or VARGRAPHIC(y)	VARGRAPHIC(z) where z = MIN(x + y, maximum length of a VARGRAPHIC)
GRAPHIC(x)	CHAR(y)* mixed data	VARGRAPHIC(z) where z = MIN(x + y, maximum length of a VARGRAPHIC)
GRAPHIC(x)	CHAR(y)* SBCS data or GRAPHIC(y)	GRAPHIC(z) where z = MIN(x + y, maximum length of a GRAPHIC)
CLOB(x)*	GRAPHIC(y) or VARGRAPHIC(y)	DBCLOB(z) where z = MIN(x + y, maximum length of a DBCLOB)
VARCHAR(x)*	GRAPHIC(y)	VARGRAPHIC(z) where z = MIN(x + y, maximum length of a VARGRAPHIC)
CLOB(x)	CHAR(y) or VARCHAR(y) or CLOB(y)	CLOB(z) where z = MIN(x + y, maximum length of a CLOB)
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where z = MIN(x + y, maximum length of a VARCHAR)
CHAR(x) mixed data	CHAR(y)	VARCHAR(z) where z = MIN(x + y, maximum length of a VARCHAR)
CHAR(x) SBCS data	CHAR(y)	CHAR(z) where z = MIN(x + y, maximum length of a CHAR)
BLOB(x)	BINARY(y) or VARBINARY(y) or BLOB(y) or CHAR(y) FOR BIT DATA or VARCHAR(y) FOR BIT DATA	BLOB(z) where z = MIN(x + y, maximum length of a BLOB)

41. Using the vertical bar (|) character might inhibit code portability between relational database products. Use the CONCAT operator in place of the || operator. Alternatively, if conformance to SQL 2003 Core standard is of primary importance, use the || operator).

Table 26. Result Data Types With Concatenation (continued)

If one operand column is ...	And the other operand is ...	The data type of the result column is ...
VARBINARY(x)	BINARY(y) or VARBINARY(y) or CHAR(y) FOR BIT DATA or VARCHAR(y) FOR BIT DATA	VARBINARY(z) where z = MIN(x + y, maximum length of a VARBINARY)
BINARY(x)	VARCHAR(y) FOR BIT DATA	VARBINARY(z) where z = MIN(x + y, maximum length of a VARBINARY)
BINARY(x)	BINARY(y) or CHAR(y) FOR BIT DATA	BINARY(z) where z = MIN(x + y, maximum length of a BINARY)
Note: * Character strings are only allowed when the other operand is a graphic string if the graphic string is Unicode.		

Table 27. Result Encoding Schemes With Concatenation

If one operand column is ...	And the other operand is ...	The data type of the result column is ...
Unicode data	Unicode data or DBCS or mixed or SBCS data	Unicode data
DBCS data	DBCS data	DBCS data
bit data	mixed or SBCS or bit data	bit data
mixed data	mixed or SBCS data	mixed data
SBCS data	SBCS data	SBCS data

If the sum of the lengths of the operands exceeds the maximum length attribute of the resulting data type:

- The length attribute of the result is the maximum length of the resulting data type.[42]
- If only blanks are truncated no warning or error occurs.
- If non-blanks are truncated, an error occurs.

If either operand can be null, the result can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first operand string followed by the second.

With mixed data this result will not have redundant shift codes “at the seam”. Thus, if the first operand is a string ending with a “shift-in” character (X'0F'), while the second operand is a character string beginning with a “shift-out” character (X'0E'), these two bytes are eliminated from the result.

42. If the expression is in the select-list, the length attribute may be further reduced in order to fit within the maximum record size. For more information, see “Maximum row sizes” on page 1027.

The actual length of the result is the sum of the lengths of the operands unless redundant shifts are eliminated; in which case, the actual length is two less than the sum of the lengths of the operands.

The CCSID of the result is determined by the CCSID of the operands as explained under "Conversion rules for operations that combine strings" on page 120. Note that as a result of these rules:

- If any operand is bit data, the result is bit data.
- If one operand is mixed data and the other is SBCS data, the result is mixed data. However, this does not necessarily mean that the result is well-formed mixed data.

Example

Concatenate the column FIRSTNME with a blank and the column LASTNAME.

```
FIRSTNME CONCAT ' ' CONCAT LASTNAME
```

Scalar fullselect

A scalar fullselect as supported in an expression is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If the select list element is an expression that is simply a column name, the result column name is based on the name of the column. Otherwise, the result column is unnamed.

See "fullselect" on page 676 for more information.

A scalar fullselect is not allowed if the query specifies:

- a distributed table,
- a table with a read trigger, or
- a logical file built over multiple physical file members.

If the scalar fullselect is a subselect, it is also referred to as a scalar subselect. See "subselect" on page 628 for more information.

Datetime operands and durations

Datetime values can be incremented, decremented, and subtracted. These operations may involve decimal numbers called *durations*. A *duration* is a positive or negative number representing an interval of time.

There are four types of durations:

Labeled durations (see diagram on page 125)

A *labeled duration* represents a specific unit of time as expressed by a number (which can be the result of an expression) followed by one of the seven duration keywords: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, or MICROSECONDS[43]. The number specified is converted as if it were assigned to a DECIMAL(15,0) number.

A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of data type DATE, TIME,

43. Note that the singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

or TIMESTAMP. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

Date duration
A *date duration* represents a number of years, months, and days, expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format *yyyymmdd*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. The result of subtracting one date value from another, as in the expression HIREDATE - BRTHDATE, is a date duration.

Time duration
A *time duration* represents a number of hours, minutes, and seconds, expressed as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format *hhmmss* where *hh* represents the number of hours, *mm* the number of minutes, and *ss* the number of seconds. The result of subtracting one time value from another is a time duration.

Timestamp duration
A *timestamp duration* represents a number of years, months, days, hours, minutes, seconds, and microseconds, expressed as a DECIMAL(20,6) number. To be properly interpreted, the number must have the format *yyyymmddhhmmss.zzzzzz*, where *yyyy, mm, dd, hh, mm, ss, and zzzzzz* represent, respectively, the number of years, months, days, hours, minutes, seconds, and microseconds. The result of subtracting one timestamp value from another is a timestamp duration.

Datetime arithmetic in SQL

The only arithmetic operations that can be performed on datetime values are addition and subtraction. If a datetime value is the operand of addition, the other operand must be a duration.

The specific rules governing the use of the addition operator with datetime values follow:

- If one operand is a date, the other operand must be a date duration or labeled duration of years, months, or days.
- If one operand is a time, the other operand must be a time duration or a labeled duration of hours, minutes, or seconds.
- If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
- Neither operand of the addition operator can be an untyped parameter marker.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow:

- If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of years, months, or days.
- If the second operand is a date, the first operand must be a date, or a string representation of a date.

- If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of hours, minutes, or seconds.
- If the second operand is a time, the first operand must be a time, or string representation of a time.
- If the first operand is a timestamp, the second operand must be a timestamp, a string representation of a timestamp, or a duration.
- If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp.
- Neither operand of the subtraction operator can be an untyped parameter marker.

Date arithmetic

Dates can be subtracted, incremented, or decremented.

Subtracting dates

The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates. The data type of the result is DECIMAL(8,0). If DATE1 is greater than or equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = DATE1 - DATE2.

```
If DAY(DATE2) <= DAY(DATE1)
    then DAY(RESULT) = DAY(DATE1) - DAY(DATE2).

If DAY(DATE2) > DAY(DATE1)
    then DAY(RESULT) = N + DAY(DATE1) - DAY(DATE2)
      where N = the last day of MONTH(DATE2).
    MONTH(DATE2) is then incremented by 1.

If MONTH(DATE2) <= MONTH(DATE1)
    then MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2).

If MONTH(DATE2) > MONTH(DATE1)
    then MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2).
    YEAR(DATE2) is then incremented by 1.

YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2).
```

For example, the result of DATE('3/15/2000') - '12/31/1999' is 215 (or, a duration of 0 years, 2 months, and 15 days).

Incrementing and decrementing dates

The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive.

If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be

February 29 of a non-leap-year. In this case, the day is changed to 28, an SQLSTATE of '01506' is assigned to the RETURNED_SQLSTATE condition area item in the SQL Diagnostics Area (or SQLWARN6 in the SQLCA is set to 'W') to indicate the end-of-month adjustment.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case, the day is set to the last day of the month, and SQLWARN6 in the SQLCA is set to 'W' to indicate the end-of-month adjustment.

Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year. Adding a labeled duration of DAYS will not cause an end-of-month adjustment.

Date durations, whether positive or negative, may also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and a warning indicator is set in the SQLCA whenever an end-of-month adjustment is necessary.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus DATE1 + X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

- DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1 - X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

- DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS

Note: If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Also note that logically equivalent expressions may not produce the same result. For example:

- (DATE('2002–01–31') + 1 MONTH) + 1 MONTH will result in a date of 2002–03–28.

does not produce the same result as

- DATE('2002–01–31') + 2 MONTHS will result in a date of 2002–03–31.

The order in which labeled date durations are added to and subtracted from dates can affect the results. For compatibility with the results of adding or subtracting date durations, a specific order must be used. When labeled date durations are added to a date, specify them in the order of YEARS + MONTHS + DAYS. When labeled date durations are subtracted from a date, specify them in the order of DAYS - MONTHS - YEARS. For example, to add one year and one day to a date, specify:

- DATE1 + 1 YEAR + 1 DAY

To subtract one year, one month, and one day from a date, specify:

- DATE1 - 1 DAY - 1 MONTH - 1 YEAR

Time arithmetic

Times can be subtracted, incremented, or decremented.

Subtracting times

The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0). If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1. If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = TIME1 - TIME2.

If SECOND(TIME2) <= SECOND(TIME1)
    then SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2).

If SECOND(TIME2) > SECOND(TIME1)
    then SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2).
    MINUTE(TIME2) is then incremented by 1.

If MINUTE(TIME2) <= MINUTE(TIME1)
    then MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2).

If MINUTE(TIME2) > MINUTE(TIME1)
    then MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2).
    HOUR(TIME2) is then incremented by 1.

HOUR(RESULT) = HOUR(TIME1) - HOUR(TIME2).

For example, the result of TIME('11:02:26') - '00:32:56' is 102930 (a duration of 10 hours, 29 minutes, and 30 seconds).

Incrementing and decrementing times

The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

Adding or subtracting a duration of seconds will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. `TIME1 + X`, where "X" is a DECIMAL(6,0) number, is equivalent to the expression:

```
TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
```

Timestamp arithmetic

Timestamps can be subtracted, incremented, or decremented.

Subtracting timestamps

The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and microseconds between the two timestamps. The data type of the result is DECIMAL(20,6). If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = TS1 - TS2.

```
If MICROSECOND(TS2) <= MICROSECOND(TS1)
      then MICROSECOND(RESULT) = MICROSECOND(TS1) -
      MICROSECOND(TS2).

If MICROSECOND(TS2) >MICROSECOND(TS1)
       then MICROSECOND(RESULT) = 1000000 +
       MICROSECOND(TS1) - MICROSECOND(TS2)
       and SECOND(TS2) is incremented by 1.
```

The seconds and minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

```
If HOUR(TS2) <= HOUR(TS1)
      then HOUR(RESULT) = HOUR(TS1) - HOUR(TS2).

If HOUR(TS2) > HOUR(TS1)
      then HOUR(RESULT) = 24 + HOUR(TS1) - HOUR(TS2)
      and DAY(TS2) is incremented by 1.
```

The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

Incrementing and decrementing timestamps

The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp, is itself a timestamp. Date and time arithmetic is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. Microseconds overflow into seconds.

Precedence of operations

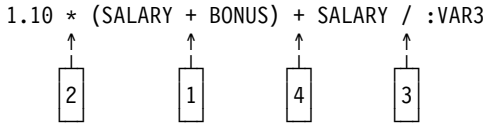
Expressions within parentheses are evaluated first. When the order of evaluation is not specified by parentheses, exponentiation is applied after prefix operators (such as -, unary minus) and before multiplication and division. Multiplication and division are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

The following table shows the priority of all operators.

Priority	Operators
1	+, - (when used for signed numeric values)
2	**
3	*, /, CONCAT, ||
4	+, - (when used between two operands)

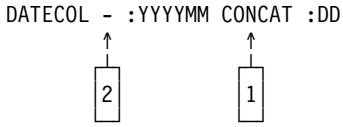
Example 1:

In this example, the first operation is the addition in (SALARY + BONUS) because it is within parenthesis. The second operation is multiplication because it is at a higher precedence level than the second addition operator and it is to the left of the division operator. The third operation is division because it is at a higher precedence level than the second addition operator. Finally, the remaining addition is performed.



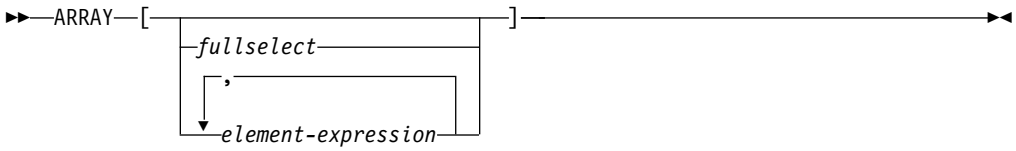
Example 2:

In this example, the first operation (CONCAT) combines the character strings in the variables YYYYMM and DD into a string representing a date. The second operation (-) then subtracts that date from the date being processed in DATECOL. The result is a date duration that indicates the time elapsed between the two dates.



ARRAY constructor

The ARRAY constructor returns an array specified by a list of expressions or a fullselect.



fullselect
A fullselect that returns a single column. The values returned by the fullselect are the elements of the array. The cardinality of the array is equal to the number of rows returned by the fullselect. An ORDER BY clause in the fullselect can be used to specify the order among the elements of the array; otherwise, the order is undefined.

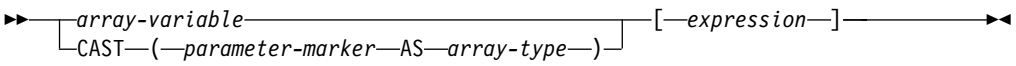
element-expression
An expression defining the value of an element within the array. The cardinality of the array is equal to the number of element expressions. The first *element-expression* is assigned to the array element with subindex 1. The second *element-expression* is assigned to the array element with subindex 2, and so on. All element expressions must have compatible data types.

If there is no expression within the brackets, the result is an empty array. The cardinality of an empty array is 0.

An ARRAY constructor can only be specified on the right side of a SET variable or *assignment-statement*.

ARRAY element specification

The ARRAY element specification returns the element from an array specified by *expression*.



array-variable
Specifies a variable or parameter of type array in an SQL procedure.

CAST(*parameter-marker* **AS** *array-type***)**
Specifies the array data type to be used for the parameter marker. The array data type passed for the parameter marker value must match this array data type exactly.

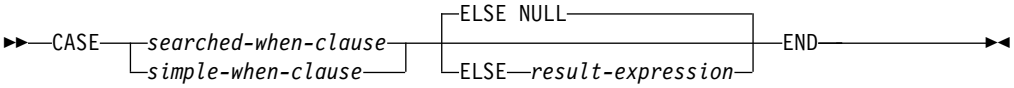
[*expression***]**
Specifies the subindex of the element that is to be extracted from the array. The subindex must return a value that is an exact numeric with zero scale or DECFLOAT. Its value must be between 1 and the maximum cardinality of the array.

The data type of the result is the data type specified for the array on the CREATE TYPE (Array) statement.

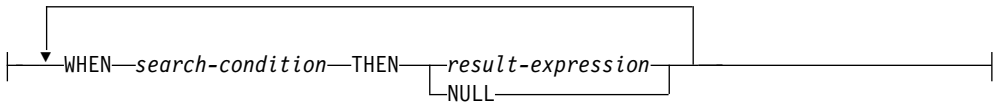
If *expression* is null or the array is null, the null value is returned.

CASE expression

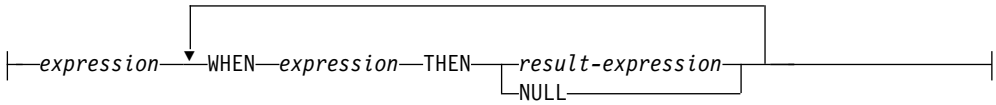
CASE expressions allow an expression to be selected based on the evaluation of one or more conditions.



searched-when-clause:



simple-when-clause:



In general, the value of the *case-expression* is the value of the *result-expression* following the first (leftmost) *when-clause* that evaluates to true. If no *when-clause* evaluates to true and the ELSE keyword is present then the result is the value of the ELSE *result-expression* or NULL. If no *when-clause* evaluates to true and the ELSE keyword is not present then the result is NULL. Note that when a *when-clause* evaluates to unknown (because of nulls), the *when-clause* is not true and hence is treated the same way as a *when-clause* that evaluates to false.

searched-when-clause
Specifies a *search-condition* that is applied to each row or group of table data presented for evaluation, and the result when that condition is true.

simple-when-clause
Specifies that the value of the *expression* prior to the first WHEN keyword is tested for equality with the value of the *expression* that follows each WHEN keyword. It also specifies the result when that condition is true.

The data type of the *expression* prior to the first WHEN keyword must be compatible with the data types of the *expression* that follows each WHEN keyword.

result-expression **or NULL**
Specifies the value that follows the THEN keyword and ELSE keywords. There must be at least one *result-expression* in the CASE expression with a defined data type. NULL cannot be specified for every case.

All *result-expressions* must have compatible data types, where the attributes of the result are determined based on the "Rules for result data types" on page 115.

search-condition
Specifies a condition that is true, false, or unknown about a row or group of table data.

The *search-condition* must not include a subquery in an EXISTS or IN predicate.

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. The following table shows the equivalent expressions using CASE or these functions.

Table 28. Equivalent CASE Expressions

CASE Expression	Equivalent Expression
CASE WHEN e1=e2 THEN NULL ELSE e1 END	NULLIF(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END	COALESCE(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE COALESCE(e2,...,eN) END	COALESCE(e1,e2,...,eN)

Examples

- If the first character of a department number is a division in the organization, then a CASE expression can be used to list the full name of the division to which each employee belongs:

```
SELECT EMPNO, LASTNAME,
  CASE SUBSTR(WORKDEPT,1,1)
  WHEN 'A' THEN 'Administration'
  WHEN 'B' THEN 'Human Resources'
  WHEN 'C' THEN 'Accounting'
  WHEN 'D' THEN 'Design'
  WHEN 'E' THEN 'Operations'
  END
FROM EMPLOYEE
```

- The number of years of education are used in the EMPLOYEE table to give the education level. A CASE expression can be used to group these and to show the level of education.

```
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
  CASE
  WHEN EDLEVEL < 15 THEN 'SECONDARY'
  WHEN EDLEVEL < 19 THEN 'COLLEGE'
  ELSE 'POST GRADUATE'
  END
FROM EMPLOYEE
```

- Another interesting example of CASE statement usage is in protecting from division by 0 errors. For example, the following code finds the employees who earn more than 25% of their income from commission, but who are not fully paid on commission:

```
SELECT EMPNO, WORKDEPT, SALARY+COMM
FROM EMPLOYEE
WHERE (CASE WHEN SALARY=0 THEN NULL
            ELSE COMM/SALARY
       END) > 0.25
```

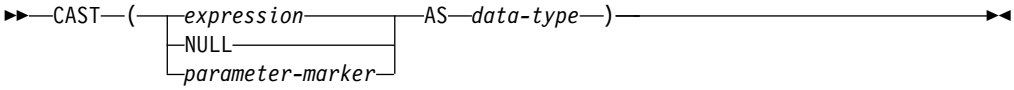
- The following CASE expressions are equivalent:

```
SELECT LASTNAME,
 CASE
 WHEN LASTNAME = 'Haas' THEN 'President'
 ...
 ELSE 'Unknown'
 END
FROM EMPLOYEE

SELECT LASTNAME,
 CASE LASTNAME
 WHEN 'Haas' THEN 'President'
```

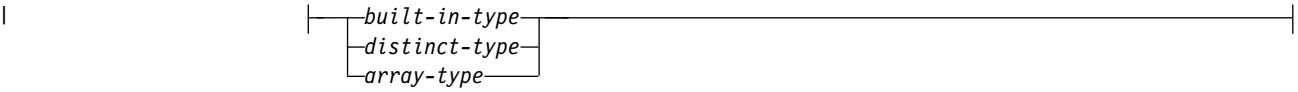
```
 ...
 ELSE 'Unknown'
 END
FROM EMPLOYEE
```

CAST specification

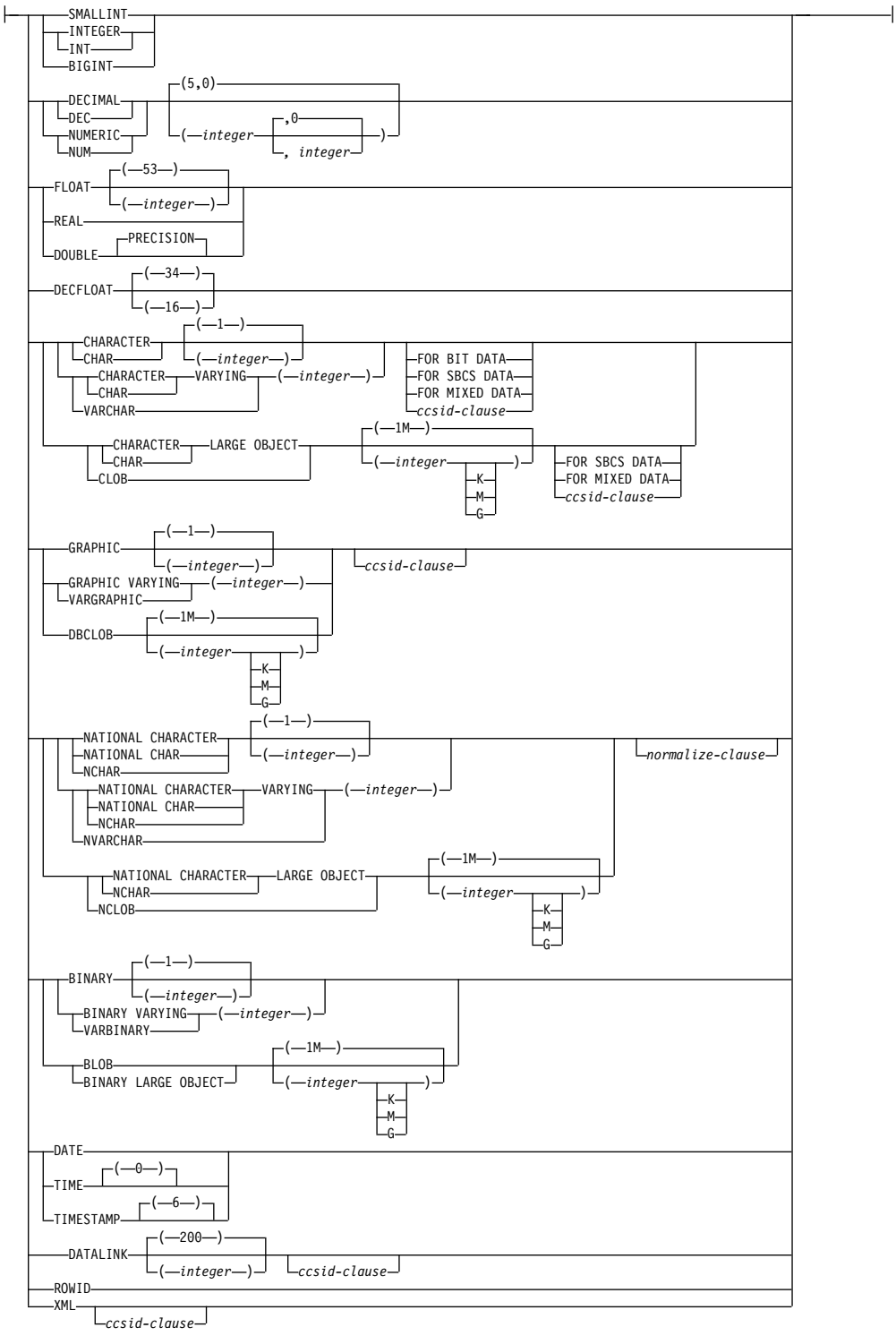
The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data-type*.



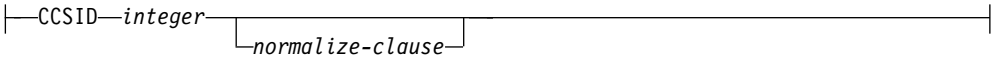
data-type:



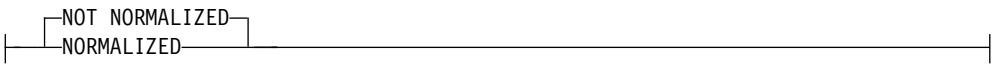
built-in-type:



ccsid-clause:



normalize-clause:



The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data-type*. If the data type of either operand is a distinct type, the privileges held by the authorization ID of the statement must include USAGE authority on the distinct type.

expression
Specifies that the cast operand is an expression other than NULL or a parameter marker. The result is the argument value converted to the specified target data type.

The supported casts are shown in "Casting between data types" on page 95, where the first column represents the data type of the cast operand (source data type) and the data types across the top represent the target data type of the CAST specification. If the cast is not supported, an error is returned.

When casting character or graphic strings to a character or graphic string with a different length, a warning is returned if truncation of other than trailing blanks occurs.

NULL
Specifies that the cast operand is the null value. The result is a null value that has the specified *data-type*.

parameter-marker
A parameter marker (specified as a question mark character) is normally considered an expression, but is documented separately in this case because it has a special meaning. If the cast operand is a *parameter-marker*, the specified *data-type* is considered a promise that the replacement will be assignable to the specified *data-type* (using storage assignment rules, see "Assignments and comparisons" on page 98). Such a parameter marker is called a *typed parameter marker*. Typed parameter markers will be treated like any other typed value for the purpose of DESCRIBE of a select list or for column assignment.

data-type
Specifies the data type of the result. If the data type is not qualified, the SQL path is used to find the appropriate data type. For more information, see "Unqualified type, variable, function, procedure, and specific names" on page 65. For a description of *data-type*, see "CREATE TABLE" on page 982. (For portability across operating systems, when specifying a floating-point data type, use REAL or DOUBLE instead of FLOAT.)

Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an *expression*, see Table 16 on page 96 for the target data types that are supported based on the data type of the cast operand.
- For a cast operand that is the keyword NULL, the target data type can be any data type.
- For a cast operand that is a parameter marker, the target data type can be any data type. If the data type is a distinct type, the application that uses the parameter marker will use the source data type of the distinct type. If the data type is an array type, the parameter marker must represent an array with a cardinality less than or equal to the maximum cardinality of the

target array data type. The data type of the parameter marker must match the data type of the target array data type exactly.

If the CCSID attribute is not specified, then:

- If the *data-type* is BINARY, VARBINARY, or BLOB, a CCSID of 65535 is used.
- If FOR BIT DATA is specified, a CCSID of 65535 is used.
- If the *expression* is a character string, and the *data-type* is CHAR, VARCHAR, or CLOB:
  - If FOR SBCS DATA is specified,
    - If the CCSID of the *expression* is a Unicode CCSID, then the single-byte CCSID associated with the default CCSID of the job is used.
    - Otherwise, the single-byte CCSID associated with the CCSID of the *expression* is used.[44]
  - If FOR MIXED DATA is specified,
    - If the CCSID of the *expression* is a Unicode CCSID, then the mixed-byte CCSID associated with the default CCSID of the job is used.
    - Otherwise, the mixed-byte CCSID associated with the CCSID of the *expression* is used.[44]
  - Otherwise, the default CCSID of the *expression* is used.[4544]
- If the *expression* is a graphic string or *expression* is a parameter marker, and the *data-type* is CHAR, VARCHAR, or CLOB:
  - If FOR SBCS DATA is specified, the single-byte CCSID associated with the default CCSID of the job is used.
  - If FOR MIXED DATA is specified, the mixed-byte CCSID associated with default CCSID of the job is used.
  - Otherwise, default CCSID of the job is used.
- If the *expression* is a character string or parameter marker, and the *data-type* is GRAPHIC, VARGRAPHIC, or DBCLOB; the CCSID 1200 is used.
- If the *expression* is a graphic string, and the *data-type* is GRAPHIC, VARGRAPHIC, or DBCLOB; the CCSID of the *expression* is used.
- If the *data-type* is XML, the CCSID value as specified by the SQL_XML_DATA_CCSID QAQQINI setting is used. See "XML Values" on page 88 for more information.
- Otherwise, the default CCSID of the job is used.

If the CCSID attribute is specified, the data will be converted to that CCSID. If NORMALIZED is specified, the data will be normalized.

For information about which casts between data types are supported and the rules for casting to a data type see "Casting between data types" on page 95.

Examples

- An application is only interested in the integer portion of the SALARY column (defined as DECIMAL(9,2)) from the EMPLOYEE table. The following CAST specification will convert the SALARY column to INTEGER.

```
SELECT EMPNO, CAST(SALARY AS INTEGER)
  FROM EMPLOYEE
```

44. For XSLTRANSFORM, if the CCSID of the *expression* is 65535, the default CCSID of the job is used.

45. If the CCSID of the *expression* is 65535, casting to CLOB is not allowed.

- Assume that two distinct types exist. T_AGE was sourced on SMALLINT and is the data type for the AGE column in the PERSONNEL table. R_YEAR was sourced on INTEGER and is the data type for the RETIRE_YEAR column in the same table. The following UPDATE statement could be prepared.

  ```
  UPDATE PERSONNEL SET RETIRE_YEAR = ?
    WHERE AGE = CAST( ? AS T_AGE )
  ```

  The first parameter is an untyped parameter marker that would have a data type of R_YEAR. An explicit CAST specification is not required in this case because the parameter marker value is assigned to the distinct type.

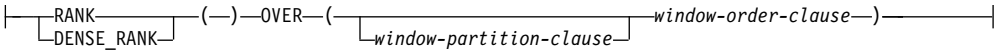
  The second parameter marker is a typed parameter marker that is cast to distinct type T_AGE. An explicit CAST specification is required in this case because the parameter marker value is compared to the distinct type.

OLAP specifications

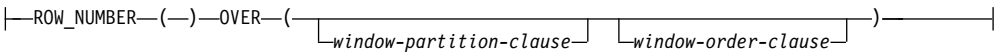
On-Line Analytical Processing (OLAP) specifications provide the ability to return ranking and row numbering as a scalar value in a query result.

OLAP-specification:

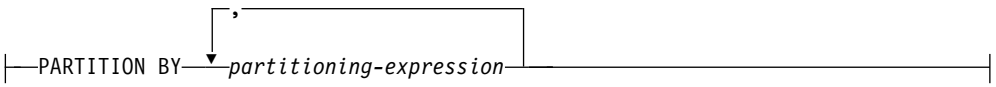
ordered-OLAP-specification:



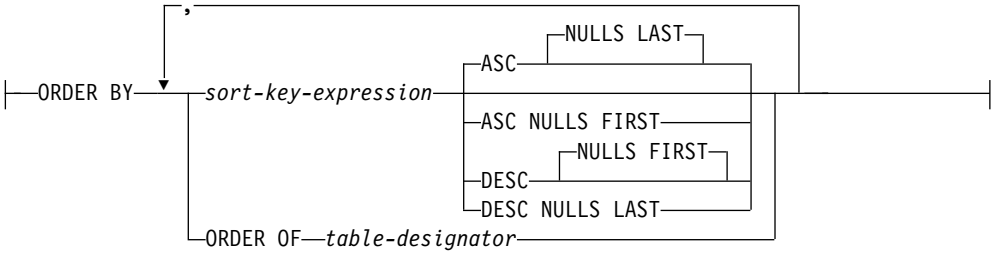
numbering-specification:



window-partition-clause:



window-order-clause:



An OLAP specification can be included in an expression in a *select-list* or the ORDER BY clause of a *select-statement*. The query result to which the OLAP specification is applied is the result table of the innermost subselect that includes the OLAP specification. OLAP specifications are sometimes referred to as *window functions*.

An OLAP specification is not valid in a WHERE, VALUES, GROUP BY, HAVING, or SET clause, or *join-condition* in an ON clause of a joined table. An OLAP specification cannot be used as an argument of an aggregate function in the *select-list*.

When invoking an OLAP specification, a window is specified that defines the rows over which the function is applied, and in what order.

The data type of the result of RANK, DENSE_RANK, or ROW_NUMBER is BIGINT. The result cannot be null.

ordered-OLAP-specification
Specifies OLAP operations that required a *window-order-clause*.

RANK or DENSE_RANK
Specifies that the ordinal rank of a row within the window is computed. Rows that are not distinct with respect to the ordering within their window are assigned the same rank. The results of ranking may be defined with or without gaps in the numbers resulting from duplicate values.

RANK
Specifies that the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering.

DENSE_RANK
Specifies that the rank of a row is defined as 1 plus the number of preceding rows that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering.

numbering-specification
Specifies an OLAP operation that returns sequential numbers for each row.

ROW_NUMBER
Specifies that a sequential row number is computed for the row within the window defined by the ordering, starting with 1 for the first row. If the ORDER BY clause is not specified in the window, the row numbers are assigned to the rows in arbitrary order, as returned by the subselect (not according to any ORDER BY clause in the *select-statement*).

window-partition-clause
Defines the partition within which the OLAP operation is applied.

PARTITION BY *(partitioning-expression,...)*
Defines the partition within which the OLAP operation is applied. A *partitioning-expression* is an expression used in defining the partitioning of the result set. Each column name referenced in a *partitioning-expression* must unambiguously reference a column of the result table of the subselect that contains the OLAP specification. A *partitioning-expression* cannot include a *scalar-fullselect* or any function that is not deterministic or has an external action.

window-order-clause
Defines the ordering of rows within a partition that is used to determine the value of the OLAP specification. It does not define the ordering of the result table.

ORDER BY *(sort-key-expression,...)*
A *sort-key-expression* is an expression used in defining the ordering of the rows within a window partition. Each column name referenced in a *sort-key-expression* must unambiguously reference a column of the result table of the subselect, including the OLAP specification. A *sort-key-expression* cannot include a *scalar-fullselect* or any function that is not deterministic or that has an external action.

The sum of the length attributes of the *sort-key-expressions* must not exceed 3.5 gigabytes.

ASC
Specifies that the values of the *sort-key-expression* are used in ascending order.

DESC
Specifies that the values of the *sort-key-expression* are used in descending order.

NULLS FIRST
Specifies that the window ordering considers null values before all non-null values in the sort order.

NULLS LAST
Specifies that the window ordering considers null values after all non-null values in the sort order.

ORDER OF *table-designator*
Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause and the table reference must identify a *nested-table-expression* or *common-table-expression*. The subselect (or fullselect) corresponding to the specified *table-designator* must include an ORDER BY clause that is dependent on the data. The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

Partitioning and ordering are performed in accordance with the comparison rules described in "Assignments and comparisons" on page 98.

If a collating sequence other than *HEX is in effect when the statement that contains the OLAP expression is executed, and the *partitioning-expressions* or the *sort-key-expressions* are SBCS data, mixed data, or Unicode data, then the results are determined using the weighted values. The weighted values are derived by applying the collating sequence to the *partitioning-expressions* and the *sort-key-expressions*.

An OLAP specification is not allowed if the query specifies:

- a distributed table,
- a table with a read trigger , or
- a logical file built over multiple physical file members.

Note: Syntax alternatives: DENSERANK can be specified in place of DENSE_RANK, and ROWNUMBER can be specified in place of ROW_NUMBER.

Examples

- Display the ranking of employees, in order by surname, according to their total salary (based on salary plus bonus) that have a total salary more than $30,000:

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
    RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
  FROM EMPLOYEE
  WHERE SALARY+BONUS > 30000
  ORDER BY LASTNAME
```

  Note that if the result is to be ordered by the ranking, then replace ORDER BY LASTNAME with:

```
ORDER BY RANK_SALARY
```

  or:

```
ORDER BY RANK() OVER (ORDER BY SALARY+BONUS DESC)
```

- Rank the departments according to their average total salary:

```
SELECT WORKDEPT, AVG(SALARY+BONUS) AS AVG_TOTAL_SALARY,
    RANK() OVER (ORDER BY AVG( SALARY+BONUS) DESC) AS RANK_AVG_SAL
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY RANK_AVG_SAL
```

- Rank the employees within a department according to their education level. Having multiple employees with the same rank in the department should not increase the next ranking value:

```
SELECT WORKDEPT, EMPNO, LASTNAME, FIRSTNME, EDLEVEL,
    DENSE_RANK() OVER (PARTITION BY WORKDEPT ORDER BY EDLEVEL DESC)
      AS RANK_EDLEVEL
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
```

- Provide row numbers in the result of a query:

```
SELECT ROW_NUMBER() OVER (ORDER BY WORKDEPT, LASTNAME ) AS NUMBER,
    LASTNAME, SALARY
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
```

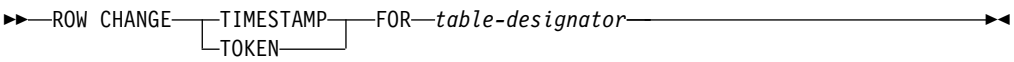
- List the top five wage earners:

```
SELECT EMPNO, LASTNAME, FIRSTNME, TOTAL_SALARY, RANK_SALARY
  FROM (SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
          RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
        FROM EMPLOYEE) AS RANKED_EMPLOYEE
  WHERE RANK_SALARY < 6
  ORDER BY RANK_SALARY
```

  Note that a nested table expression was used to first compute the result, including the rankings, before the rank could be used in the WHERE clause. A common table expression could also have been used.

ROW CHANGE expression

A ROW CHANGE expression returns a token or a timestamp that represents the last change to a row.



ROW CHANGE TIMESTAMP

Specifies that a timestamp is returned that represents the last time when a row was changed. If the row has not been changed, the result is the time that the initial value was inserted. If the table does not have a row change timestamp, this expression is not allowed.

ROW CHANGE TOKEN

Specifies that a token that is a BIGINT value is returned that represents a relative point in the modification sequence of a row. If the row has not been changed, the result is a token that represents when the initial value was inserted.

FOR *table-designator*

Specifies a table designator of the subselect. For more information about table designators, see "Table designators" on page 143. In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified. The table designator cannot identify a table function or a *data-change-table-reference*. If the table designator identifies a view or a nested table expression, the expression returns the ROW CHANGE TOKEN or ROW CHANGE TIMESTAMP of its base table.

The result can be the null value. These expressions are not deterministic.

Example

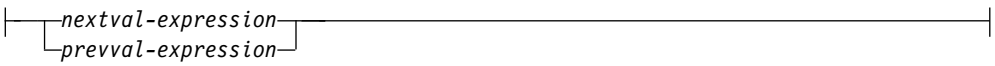
- Find all rows that have been changed in the last day:

```
SELECT *
  FROM  ORDERS
  WHERE ROW CHANGE TIMESTAMP FOR  ORDERS > CURRENT TIMESTAMP  - 24 HOURS
```

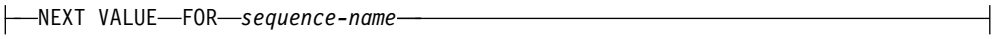
Sequence reference

A sequence is referenced by using the NEXT VALUE and PREVIOUS VALUE expressions specifying the name of the sequence.

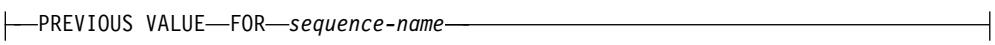
sequence-reference:



nextval-expression:



prevval-expression:



A sequence is referenced by using the NEXT VALUE and PREVIOUS VALUE expressions specifying the name of the sequence.

nextval-expression
A NEXT VALUE expression generates and returns the next value for a specified sequence. A new value is generated for a sequence when a NEXT VALUE expression specifies the name of the sequence. However, if there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the sequence value is incremented only once for each row of the result, and all instances of NEXT VALUE return the same value for a row of the result. NEXT VALUE is a non-deterministic expression with external actions since it causes the sequence value to be incremented.

When the next value for the sequence is generated, if the maximum value for an ascending sequence or the minimum value for a descending sequence of the logical range of the sequence is exceeded and the NO CYCLE option is in effect, then an error is returned.

The data type and length attributes of the result of a NEXT VALUE expression are the same as for the specified sequence. The result cannot be null.

prevval-expression
A PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous statement within the current application process. This value can be repeatedly referenced by using PREVIOUS VALUE expressions and specifying the name of the sequence. There may be multiple instances of PREVIOUS VALUE expressions specifying the same sequence name within a single statement and they all return the same value.

A PREVIOUS VALUE expression can be used only if a NEXT VALUE expression specifying the same sequence name has already been referenced in the current application process.

The data type and length attributes of the result of a PREVIOUS VALUE expression are the same as for the specified sequence. The result cannot be null.

sequence-name
Identifies the sequence to be referenced. The *sequence-name* must identify a sequence that exists at the current server.

Notes

Authorization: If a sequence is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the sequence identified in the statement,
  - The USAGE privilege on the sequence, and
  - The system authority *EXECUTE on the library containing the sequence
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Sequence.

Generating values with NEXT VALUE: When a value is generated for a sequence, that value is consumed, and the next time that a value is requested, a new value will be generated. This is true even when the statement containing the NEXT VALUE expression fails or is rolled back.

Scope of PREVIOUS VALUE: The PREVIOUS VALUE value persists until the next value is generated for the sequence in the current session, the sequence is dropped or altered, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements. The value of PREVIOUS VALUE cannot be directly set and is a result of executing the NEXT VALUE expression for the sequence.

A technique commonly used, especially for performance, is for an application or product to manage a set of connections and route transactions to an arbitrary connection. In these situations, the availability of the PREVIOUS VALUE for a sequence should only be relied on until the end of the transaction.

Use as a Unique Key Value: The same sequence number can be used as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row (this generates the sequence value), and a PREVIOUS VALUE expression for the other rows (the instance of PREVIOUS VALUE refers to the sequence value most recently generated in the current session), as shown below:

```
INSERT INTO ORDER (ORDERNO, CUSTNO)
  VALUES (NEXT VALUE FOR ORDER_SEQ, 123456)

INSERT INTO LINE_ITEM (ORDERNO, PARTNO, QUANTITY)
  VALUES (PREVIOUS VALUE FOR ORDER_SEQ, 987654, 1)
```

Allowed use of NEXT VALUE and PREVIOUS VALUE: NEXT VALUE and PREVIOUS VALUE expressions can be specified in the following places:

- Within the *select-clause* of a SELECT statement or SELECT INTO statement as long as the statement does not contain a DISTINCT keyword, a GROUP BY clause, an ORDER BY clause, a UNION keyword, an INTERSECT keyword, or EXCEPT keyword
- Within a VALUES clause of a *fullselect* (NEXT VALUE is not allowed)
- Within a VALUES clause of an INSERT statement
- Within the *select-clause* of the fullselect of an INSERT statement
- Within the SET clause of a searched or positioned UPDATE statement, though NEXT VALUE cannot be specified in the *select-clause* of the subselect of an expression in the SET clause

  A PREVIOUS VALUE expression can be specified anywhere within a SET clause of an UPDATE statement, but a NEXT VALUE expression can be specified only

in a SET clause if it is not within the *select-clause* of the fullselect of an expression. For example, the following uses of sequence expressions are supported:

```
UPDATE T SET C1 = (SELECT PREVIOUS VALUE FOR S1 FROM T)

UPDATE T SET C1 = PREVIOUS VALUE FOR S1

UPDATE T SET C1 = NEXT VALUE FOR S1
```

The following use of a sequence expression is not supported:

```
UPDATE T SET C1 = (SELECT NEXT VALUE FOR S1 FROM T)
```

- Within an *assignment-statement*, except within the *select-clause* of the fullselect of an expression. The following uses of sequence expressions are supported:

  ```
  SET :ORDERNUM = NEXT VALUE FOR INVOICE

  SET :ORDERNUM = PREVIOUS VALUE FOR INVOICE
  ```

  The following use of a sequence expression is not supported:

  ```
  SET :X = (SELECT NEXT VALUE FOR S1 FROM T)

  SET :X = (SELECT PREVIOUS VALUE FOR S1 FROM T)
  ```

- Within a VALUES or VALUES INTO statement though not within the *select-clause* of the fullselect of an expression
- Within the *SQL-routine-body* of a CREATE PROCEDURE statement
- Within the *SQL-trigger-body* of a CREATE TRIGGER statement (PREVIOUS VALUE is not allowed)
- Within the argument list of a CALL statement.
- Within a default expression for CREATE PROCEDURE.

Restrictions on the use of NEXT VALUE and PREVIOUS VALUE: NEXT VALUE and PREVIOUS VALUE expressions cannot be specified in the following places:

- Within a materialized query table definition in a CREATE TABLE or ALTER TABLE statement
- Within a CHECK constraint
- Within a view definition
- Within a CREATE INDEX statement
- Within the *SQL-routine-body* of a CREATE FUNCTION statement

In addition, the NEXT VALUE expression cannot be specified in the following places:

- CASE expression
- Parameter list of an aggregate function
- Subquery in a context other than those explicitly allowed
- SELECT statement for which the outer SELECT contains a DISTINCT operator or a GROUP BY clause
- SELECT statement for which the outer SELECT is combined with another SELECT statement using the UNION, INTERSECT, or EXCEPT operator
- SELECT statement that contains an OFFSET clause.
- Join condition of a join
- Nested table expression
- Parameter list of a table function

- *select-clause* of the fullselect of an expression in the SET clause of an UPDATE statement
- WHERE clause of the outermost SELECT statement or a DELETE, or UPDATE statement
- ORDER BY clause of the outermost SELECT statement
- IF, WHILE, DO . . . UNTIL, or CASE statements in an SQL routine

Using sequence expressions with a cursor: Normally, a SELECT NEXT VALUE FOR ORDER_SEQ FROM T1 would produce a result table containing as many generated values from the sequence ORDER_SEQ as the number of rows retrieved from T1. A reference to a NEXT VALUE expression in the SELECT statement of a cursor refers to a value that is generated for a row of the result table. A sequence value is generated for a NEXT VALUE expression each time a row is retrieved.

If blocking is done at a client in a DRDA environment, sequence values may get generated at the DB2 server before the processing of an application's FETCH statement. If the client application does not explicitly FETCH all the rows that have been retrieved from the database, the application will never see all those generated values of the sequence (as many as the rows that were not FETCHed). These values may constitute a gap in the sequence.

A reference to the PREVIOUS VALUE expression in a SELECT statement of a cursor is evaluated at OPEN time. In other words, a reference to the PREVIOUS VALUE expression in the SELECT statement of a cursor refers to the last value generated by this application process for the specified sequence prior to the opening of the cursor. Once evaluated at OPEN time, the value returned by PREVIOUS VALUE within the body of the cursor will not change from FETCH to FETCH, even if NEXT VALUE is invoked within the body of the cursor. After the cursor is closed, the value of PREVIOUS VALUE will be the last NEXT VALUE generated by the application process.

Syntax alternatives: The keywords NEXTVAL and PREVVAL can be used as alternatives for NEXT VALUE and PREVIOUS VALUE respectively.

Examples

- Assume that there is a table called ORDER, and that a sequence called ORDER_SEQ is created as follows:

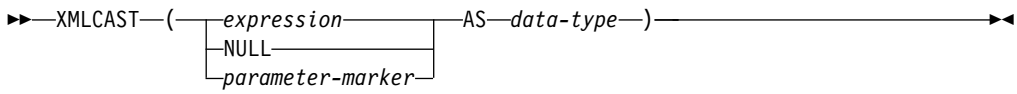
```
CREATE SEQUENCE ORDER_SEQ
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 24
```

  Following are some examples of how to generate an ORDER_SEQ sequence number with a NEXT VALUE expression:

```
INSERT INTO ORDER (ORDERNO, CUSTNO)
  VALUES (NEXT VALUE FOR ORDER_SEQ, 123456)

UPDATE ORDER
  SET ORDERNO = NEXT VALUE FOR ORDER_SEQ
  WHERE CUSTNO = 123456

VALUES NEXT VALUE FOR ORDER
  INTO :HV_SEQ
```

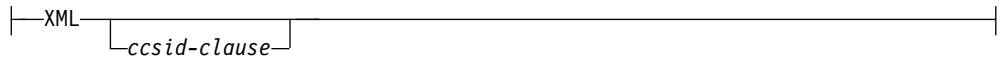
XMLCAST specification

The XMLCAST specification returns the cast operand (the first operand) cast to the XML *data-type*.

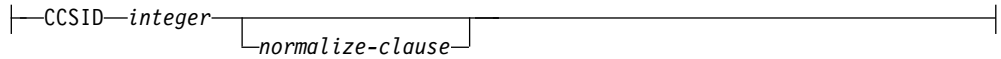


data-type:

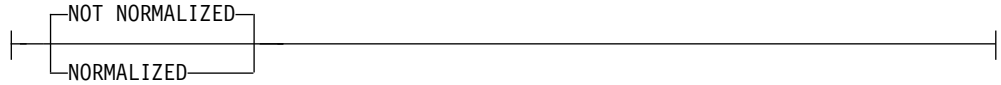
built-in-type:



ccsid-clause:



normalize-clause:



expression
Specifies that the cast operand is an expression other than NULL or a parameter marker. It must have a data type of XML. The result is the argument value converted to the specified XML target data type and CCSID.

NULL
Specifies that the cast operand is the null value. The result is a null value that has the XML data type.

parameter-marker
A parameter marker (specified as a question mark character) is normally considered an expression, but is documented separately in this case because it has a special meaning. If the cast operand is a *parameter-marker*, the XML data type is considered a promise that the replacement will be assignable to the XML data type. Such a parameter marker is called a *typed parameter marker*. Typed parameter markers will be treated like any other typed value for the purpose of DESCRIBE of a select list or for column assignment.

data-type
Specifies the data type of the result. The data type must be XML or a distinct type based on XML.

If the CCSID attribute is not specified, then the CCSID value as specified by the SQL_XML_DATA_CCSID QAQQINI setting is used. See "XML Values" on page 88 for more information.

If the CCSID attribute is specified, the data will be converted to that CCSID. If NORMALIZED is specified, the data will be normalized.

Example

- Create a null XML value:

```
VALUES(XMLCAST(NULL AS XML))
```

Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given value, row, or group.

The following rules apply to all types of predicates:

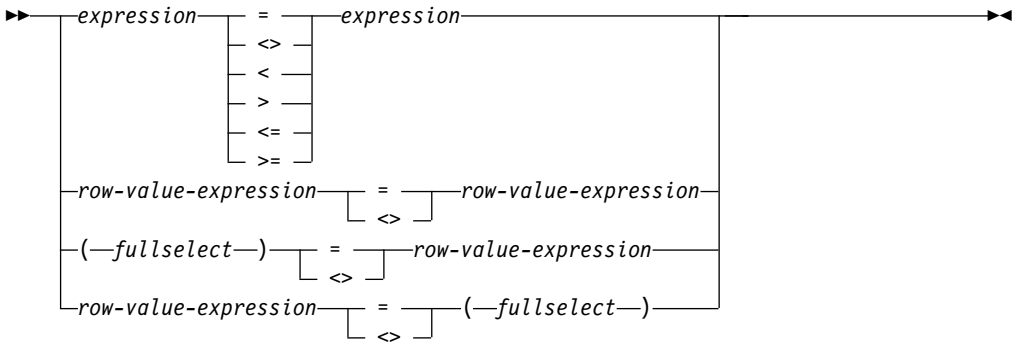
- Predicates are evaluated after the expressions that are operands of the predicate.
- All values specified in the same predicate must be compatible.
- The value of a variable may be null (that is, the variable may have a negative indicator variable).
- The CCSID conversion of operands of predicates involving two or more operands are done according to "Conversion rules for comparison:" on page 112.
- Use of a DataLink value is limited to the NULL predicate.

Row-value expression: The operand of several predicates (basic, quantified, and IN) can be a *row-value-expression*:

A *row-value-expression* returns a single row that consists of one or more column values. The values can be specified as a list of expressions. The number of columns that are returned by the *row-value-expression* is equal to the number of expressions that are specified in the list.

Basic predicate

A *basic predicate* compares two values or compares a set of values with another set of values.



Notes:

[1] Other comparison operators are also supported.[46]

When a single *expression* is specified on the left side of the operator, another *expression* must be specified on the right side. The data types of the corresponding expressions must be compatible. The value of the expression on the left side is compared with the value of the expression on the right side. If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

When a *row-value-expression* is specified on the left side of the operator (= or <>) and another *row-value-expression* is specified on the right side of the operator, both *row-value-expressions* must have the same number of value expressions. The data types of the corresponding expressions of the *row-value-expressions* must be compatible. The value of each expression on the left side is compared with the value of its corresponding expression on the right side.

When a *row-value-expression* is specified and a *fullselect* is also specified:

- SELECT * is not allowed in the outermost select lists of the *fullselect*.
- The result table of the *fullselect* must have the same number of columns as the *row-value-expression*. The data types of the corresponding expressions of the *row-value-expression* and the *fullselect* must be compatible. The value of each expression on the left side is compared with the value of its corresponding expression on the right side.

The result of the predicate depends on the operator:

- If the operator is =, the result of the predicate is:
  - True if all pairs of corresponding value expressions evaluate to true.

46. The following forms of the comparison operators are also supported in basic and quantified predicates: !=, !<, !>, ¬=,¬<, and¬> are supported. All these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators and are not recommended for use when writing new SQL statements.

    Some keyboards must use the hex values for the not (¬) symbol. The hex value varies and is dependent on the keyboard that is used. A not sign (¬) or the character that must be used in its place in certain countries or regions, can cause parsing errors in statements passed from one database server to another. The problem occurs if the statement undergoes character conversion with certain combinations of source and target CCSIDs. To avoid this problem, substitute an equivalent operator for any operator that includes a not sign. For example, substitute '<>' for '¬=', '<=' for '¬>', and '>=' for '¬<'.

  - False if any one pair of corresponding value expressions evaluates to false.
  - Otherwise, unknown (that is, if at least one comparison of corresponding value expressions is unknown because of a null value and no pair of corresponding value expressions evaluates to false).
- If the operator is <>, the result of the predicate is:
  - True if any one pair of corresponding value expressions evaluates to false.
  - False if all pairs of corresponding value expressions evaluate to true.
  - Otherwise, unknown (that is, if at least one comparison of corresponding value expressions is unknown because of a null value and no pair of corresponding value expressions evaluates to true).

If the corresponding operands of the predicate are SBCS data, mixed data, or Unicode data, and if the collating sequence in effect at the time the statement is executed is not *HEX, then the comparison of the operands is performed using weighted values for the operands. The weighted values are based on the collating sequence.

For values *x* and *y*:

Predicate	**Is true if and only if...**
x **=** *y*	*x* is equal to *y*
*x***<>** *y*	*x* is not equal to *y*
x **<** *y*	*x* is less than *y*
x **>** *y*	*x* is greater than *y*
*x***>=** *y*	*x* is greater than or equal to *y*
*x***<=** *y*	*x* is less than or equal to *y*

Examples

Example 1

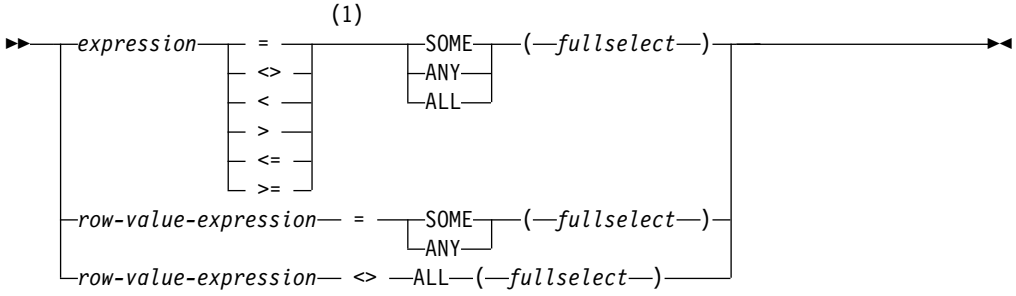
```
EMPNO = '528671'

PRTSTAFF <> :VAR1

SALARY + BONUS + COMM < 20000

SALARY > (SELECT AVG(SALARY)
          FROM EMPLOYEE)
```

Example 2: List the name, first name, and salary of the employee who is responsible for the 'OP1000' project.

```
SELECT  LASTNAME, FIRSTNME, SALARY
  FROM  EMPLOYEE X
  WHERE  EMPNO = ( SELECT  RESPEMP
                   FROM PROJA1 Y
                   WHERE MAJPROJ = 'OP1000' )
```

Quantified predicate

A *quantified predicate* compares a value or values with a set of values.



Notes:

1 Other comparison operators are also supported. [46]

When *expression* is specified, the fullselect must return a single result column. The fullselect can return any number of values, whether null or not null. The result depends on the operator that is specified:

- When ALL is specified, the result of the predicate is:
  - True if the result of the fullselect is empty, or if the specified relationship is true for every value returned by the fullselect.
  - False if the specified relationship is false for at least one value returned by the fullselect.
  - Unknown if the specified relationship is not false for any values returned by the fullselect and at least one comparison is unknown because of a null value.
- When SOME or ANY is specified, the result of the predicate is:
  - True if the specified relationship is true for at least one value returned by the fullselect.
  - False if the result of the fullselect is empty, or if the specified relationship is false for every value returned by the fullselect.
  - Unknown if the specified relationship is not true for any of the values returned by the fullselect and at least one comparison is unknown because of a null value.

When *row-value-expression* is specified, the number of result columns returned by the fullselect must be the same as the number of value expressions specified by *row-value-expression*. The fullselect can return any number of rows of values. The data types of the corresponding expressions of the row value expressions must be compatible. The value of each expression from *row-value-expression* is compared with the value of the corresponding result column from the fullselect. SELECT * is not allowed in the outermost select lists of the *fullselect*. The value of the predicate depends on the operator that is specified:

- When ALL is specified, the result of the predicate is:
  - True if the result of the fullselect is empty or if the specified relationship is true for every row returned by fullselect.
  - False if the specified relationship is false for at least one row returned by the fullselect.
  - Unknown if the specified relationship is not false for any row returned by the fullselect and at least one comparison is unknown because of a null value.
- When SOME or ANY is specified, the result of the predicate is:

- True if the specified relationship is true for at least one row returned by the fullselect.
- False if the result of the fullselect is empty or if the specified relationship is false for every row returned by the fullselect.
- Unknown if the specified relationship is not true for any of the rows returned by the fullselect and at least one comparison is unknown because of a null value.

If the corresponding operands of the predicate are SBCS data, mixed data, or Unicode data, and if the collating sequence in effect at the time the statement is executed is not *HEX, then the comparison of the operands is performed using weighted values for the operands. The weighted values are based on the collating sequence.

Examples

Table **TBLA**

```
COLA
-----
    1
    2
    3
    4
 null
```

Table **TBLB**

```
COLB
-----
    2
    3
```

Example 1

```
SELECT * FROM TBLA WHERE COLA =  ANY(SELECT COLB FROM TBLB)
```

Results in 2,3. The fullselect returns (2,3). COLA in rows 2 and 3 equals at least one of these values.

Example 2

```
SELECT * FROM TBLA WHERE COLA > ANY(SELECT COLB FROM TBLB)
```

Results in 3,4. The fullselect returns (2,3). COLA in rows 3 and 4 is greater than at least one of these values.

Example 3

```
SELECT * FROM TBLA WHERE COLA > ALL(SELECT COLB FROM TBLB)
```

Results in 4. The fullselect returns (2,3). COLA in row 4 is the only one that is greater than both these values.

Example 4

```
SELECT * FROM TBLA WHERE COLA > ALL(SELECT COLB FROM TBLB WHERE COLB<0)
```

Results in 1,2,3,4 and null. The fullselect returns no values. Thus, the result of the predicate is true for all rows in TBLA.

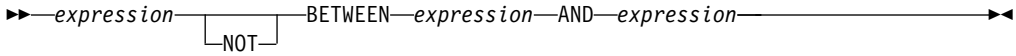
Example 5

```
SELECT * FROM TBLA WHERE COLA > ANY(SELECT COLB FROM TBLB WHERE COLB<0)
```

Results in the empty set. The fullselect returns no values. Thus, the result of the predicate is false for all rows in TBLA.

BETWEEN predicate

The BETWEEN predicate compares a value with a range of values.



If the data types of the operands are not the same, all values are converted to the data type that would result by applying the "Rules for result data types" on page 115.

If the operands of the BETWEEN predicate are strings with different CCSIDs, operands are converted as if the below logically-equivalent search conditions were specified.

The BETWEEN predicate:

```
value1 BETWEEN value2 AND value3
```

is logically equivalent to the search condition:

```
value1 >= value2 AND value1 <= value3
```

The BETWEEN predicate:

```
value1 NOT BETWEEN value2 AND value3
```

is logically equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3)
```

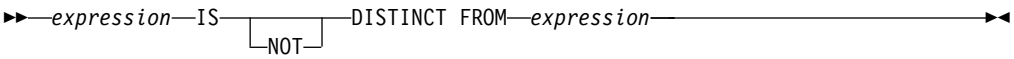
If the operands of the predicate are SBCS data, mixed data, or Unicode data, and if the collating sequence in effect at the time the statement is executed is not *HEX, then the comparison of the operands is performed using weighted values for the operands. The weighted values are based on the collating sequence.

Examples

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000

SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

DISTINCT predicate

The DISTINCT predicate compares a value with another value.



When the predicate is IS DISTINCT, the result of the predicate is true if the comparison of the expressions evaluates to true. Otherwise, the result of the predicate is false. The result cannot be unknown.

When the predicate IS NOT DISTINCT FROM, the result of the predicate is true if the comparison of the expressions evaluates to true (null values are considered equal to null values). Otherwise, the result of the predicate is false. The result cannot be unknown.

The DISTINCT predicate:

```
value1 IS NOT DISTINCT FROM value2
```

is logically equivalent to the search condition:

```
( value1 IS NOT NULL AND value2 IS NOT NULL AND value1 = value2 )
      OR
( value1 IS NULL AND value2 IS NULL )
```

The DISTINCT predicate:

```
value1 IS DISTINCT FROM value2
```

is logically equivalent to the search condition:

```
NOT (value1 IS NOT DISTINCT FROM  value2)
```

If the operands of the DISTINCT predicate are strings with different CCSIDs, operands are converted as if the above logically-equivalent search conditions were specified.

If the operands of the predicate are SBCS data, mixed data, or Unicode data, and if the collating sequence in effect at the time the statement is executed is not *HEX, then the comparison of the operands is performed using weighted values for the operands. The weighted values are based on the collating sequence.

Example

Assume that table T1 exists and it has a single column C1, and three rows with the following values for C1: 1, 2, null. The following query produces the following results:

```
SELECT * FROM T1
   WHERE C1 IS DISTINCT FROM :HV
```

C1	:HV	Result
1	2	True
2	2	False
1	Null	True
Null	Null	False

The following query produces the following results:

```
SELECT * FROM T1
   WHERE C1 IS NOT DISTINCT FROM :HV
```

C1	:HV	Result
1	2	False
2	2	True
1	Null	False
Null	Null	True

EXISTS predicate

The EXISTS predicate tests for the existence of certain rows.

The fullselect may specify any number of columns, and

- The result is true only if the number of rows specified by the fullselect is not zero.
- The result is false only if the number of rows specified by the fullselect is zero.
- The result cannot be unknown.

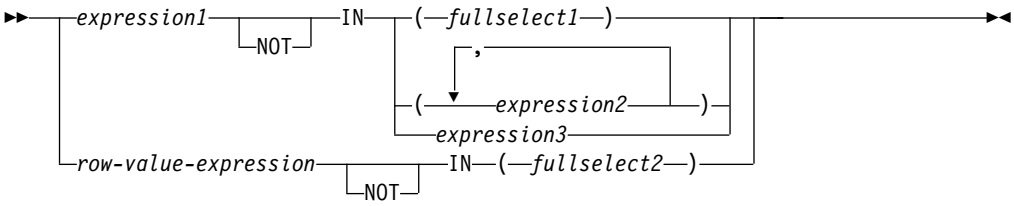
The values returned by the fullselect are ignored.

Example

```
EXISTS (SELECT *
          FROM EMPLOYEE WHERE SALARY > 60000)
```

IN predicate

The IN predicate compares a value or values with a set of values.



When *expression1* is specified, the IN predicate compares a value with a set of values. When *fullselect1* is specified, the fullselect must return a single result column, and can return any number of values, whether null or not null. The data type of *expression1* and the data type of the result column of the *fullselect1*, *expression2*, or *expression3* must be compatible. Each variable must identify a structure or variable that is described in accordance with the rule for declaring host structures or variables.

When a *row-value-expression* is specified, the IN predicate compares values with a collection of values.

- SELECT * is not allowed in the outermost select list of *fullselect2*.
- The result table of the *fullselect2* must have the same number of columns as *row-value-expression*. The data types of the corresponding expressions of *row-value-expression* and of its the corresponding result column of *fullselect2* must be compatible. The value of each expression in *row-value-expression* is compared with the value of its corresponding result column of *fullselect2*.

The value of the predicate depends on the operator that is specified:

- When the operator is IN, the result of the predicate is:
  - True if at least one row returned from the *fullselect2* is equal to the *row-value-expression*.
  - False if the result of *fullselect2* is empty or if no row returned from the *fullselect2* is equal to the *row-value-expression*.
  - Otherwise, unknown (that is, if the comparison of *row-value-expression* to the row returned from *fullselect2* evaluates to unknown because of a null value for at least one row returned from *fullselect2* and no row returned from *fullselect2* is equal to the *row-value-expression*).
- When the operator is NOT IN, the result of the predicate is:
  - True if the result of *fullselect2* is empty or if the *row-value-expression* is not equal to any of the rows returned by *fullselect2*.
  - False if the *row-value-expression* is equal to at least one row returned by *fullselect2*.
  - Otherwise, unknown (that is, if the comparison of *row-value-expression* to the row returned from *fullselect2* evaluates to unknown because of a null value for at least one row returned from *fullselect2* and the comparison of *row-value-expression* to the row returned from *fullselect2* is not true for any row returned by *fullselect2*).

An IN predicate is equivalent to other predicates as follows:

IN predicate	Equivalent predicate
expression **IN (**expression**)**	expression **=** expression
expression **IN (**fullselect**)**	expression **= ANY (**fullselect**)**
expression **NOT IN (**fullselect**)**	expression **<> ALL (**fullselect**)**

expression **IN (**expression1, expression2, ..., expressionn**)**	expression **IN (SELECT** * **FROM** R**)**
	
	Where T is a table with a single row and R is a temporary table formed by the following fullselect:
	`SELECT expression1 FROM T`
	`  UNION`
	`SELECT expression2 FROM T`
	`  UNION`
	`  .`
	`  .`
	`  .`
	`  UNION`
	`SELECT expressionn FROM T`

row-value-expression **IN (**fullselect**)**	row-value-expression **= SOME (** fullselect**)**
row-value-expression **IN (**fullselect**)**	row-value-expression **= ANY (** fullselect**)**
row-value-expression **NOT IN (**fullselect**)**	row-value-expression **<> ALL (** fullselect**)**

If the operands of the IN predicate have different data types or attributes, the rules used to determine the data type for evaluation of the IN predicate are those for UNION, UNION ALL, EXCEPT, and INTERSECT. For a description, see "Rules for result data types" on page 115.

If the operands of the IN predicate are strings with different CCSIDs, the rules used to determine which operands are converted are those for operations that combine strings. For a description, see "Conversion rules for operations that combine strings" on page 120.

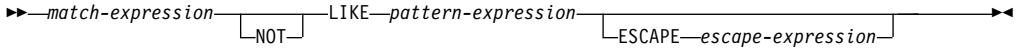
If the corresponding operands of the predicate are SBCS data, mixed data, or Unicode data, and if the collating sequence in effect at the time the statement is executed is not *HEX, then the comparison of the operands is performed using weighted values for the operands. The weighted values are based on the collating sequence.

Examples

```
DEPTNO IN ('D01', 'B01', 'C01')

EMPNO IN(SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

LIKE predicate

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and percent sign have special meanings. Trailing blanks in a pattern are a part of the pattern.



If the value of any of the arguments is null, the result of the LIKE predicate is unknown.

The *match-expression*, *pattern-expression*, and *escape-expression* must identify strings or numbers. A numeric argument is cast to a character string before evaluating the predicate. For more information about converting numeric to a character string, see "VARCHAR" on page 531. The values for *match-expression*, *pattern-expression*, and *escape-expression* must either all be binary strings or none can be binary strings. The three arguments can include a mixture of character strings and graphic strings.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source type.

If the operands of the predicate are SBCS data, mixed data, or Unicode data, and if the collating sequence in effect at the time the statement is executed is not *HEX, then the comparison of the operands is performed using weighted values for the operands. The weighted values are based on the collating sequence. An ICU collating sequence is not allowed with a LIKE predicate.

With character strings, the terms *character*, *percent sign*, and *underscore* in the following discussion refer to single-byte characters. With graphic strings, the terms refer to double-byte or Unicode characters. With binary strings, the terms refer to the code points of those single-byte characters.

match-expression
: An expression that specifies the string that is to be examined to see if it conforms to a certain pattern of characters.

LIKE *pattern-expression*
: An expression that specifies the string that is to be matched.

  Simple description: A simple description of the LIKE pattern is as follows:

  - The underscore sign (_) represents any single character.
  - The percent sign (%) represents a string of zero or more characters.
  - Any other character represents itself.

  If the *pattern-expression* needs to include either the underscore or the percent character, the *escape-expression* is used to specify a character to precede either the underscore or percent character in the pattern.

  Rigorous description: Let *x* denote a value of *match-expression* and *y* denote the value of *pattern-expression*.

  The string *y* is interpreted as a sequence of the minimum number of substring specifiers so each character of *y* is part of exactly one substring specifier. A substring specifier is an underscore, a percent sign, or any nonempty sequence of characters other than an underscore or a percent sign.

The result of the predicate is unknown if *x* or *y* is the null value. Otherwise, the result is either true or false. The result is true if *x* and *y* are both empty strings or if there exists a partitioning of *x* into substrings such that:

- A substring of *x* is a sequence of zero or more contiguous characters and each character of *x* is part of exactly one substring.
- If the *n*th substring specifier is an underscore, the *n*th substring of *x* is any single character.
- If the *n*th substring specifier is a percent sign, the *n*th substring of *x* is any sequence of zero or more characters.
- If the *n*th substring specifier is neither an underscore nor a percent sign, the *n*th substring of *x* is equal to that substring specifier and has the same length as that substring specifier.
- The number of substrings of *x* is the same as the number of substring specifiers.

It follows that if *y* is an empty string and *x* is not an empty string, the result is false. Similarly, it follows that if *x* is an empty string and *y* is not an empty string consisting of other than percent signs, the result is false.

The predicate *x* `NOT LIKE` *y* is equivalent to the search condition `NOT(`*x* `LIKE` *y*`)`.

If necessary, the CCSID of the *match-expression*, *pattern-expression*, and *escape-expression* are converted to the compatible CCSID between the *match-expression* and *pattern-expression*.

Mixed data: If the column is mixed data, the pattern can include both SBCS and DBCS characters. The special characters in the pattern are interpreted as follows:

- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one DBCS character.
- A percent sign (either SBCS or DBCS) refers to any number of characters of any type, either SBCS or DBCS.
- Redundant shifts in *match-expression* and *pattern-expression* are ignored.[47]

Unicode data: For Unicode, the special characters in the pattern are interpreted as follows:

- An SBCS or DBCS underscore refers to one character (a character can be one or more bytes)
- A percent sign (either SBCS or DBCS) refers to a string of zero or more characters (a character can be one or more bytes).

When the LIKE predicate is used with Unicode data, the Unicode percent sign and underscore use the code points indicated in the following table:

Character	UTF-8	UTF-16 or UCS-2
Half-width %	X'25'	X'0025'
Full-width %	X'EFBC85'	X'FF05'
Half-width _	X'5F'	X'005F'
Full-width _	X'EFBCBF'	X'FF3F'

47. Redundant shifts are normally ignored. To guarantee that they are ignored, however, specify the IGNORE_LIKE_REDUNDANT_SHIFTS query attribute. See Database Performance and Query Optimization for information about setting query attributes.

The full-width or half-width % matches zero or more characters. The full-width or half width _ character matches exactly one character. (For EBCDIC data, a full-width _ character matches one DBCS character.)

Binary data: If the column is binary data, the pattern contains bytes. The special bytes in the pattern are interpreted as follows:

- The code point for an SBCS underscore (X'6D') refers to one byte.
- The code point for an SBCS percent (X'6C') refers to any number of bytes.

Parameter marker:

When the pattern specified in a LIKE predicate is a parameter marker, and a fixed-length character variable is used to replace the parameter marker; specify a value for the variable that is the correct length. If a correct length is not specified, the select will not return the intended results.

For example, if the variable is defined as CHAR(10), and the value WYSE% is assigned to that variable, the variable is padded with blanks on assignment. The pattern used is

```
'WYSE%     '
```

This pattern requests the database manager to search for all values that start with WYSE and end with five blank spaces. If you intended to search for only the values that start with 'WYSE' you should assign the value 'WYSE%%%%%%' to the variable.

ESCAPE *escape-expression*

An expression that specifies a character to be used to modify the special meaning of the underscore (_) and percent (%) characters in the pattern-expression. This allows the LIKE predicate to be used to match values that contain the actual percent and underscore characters. The following rules apply the use of the ESCAPE clause and the *escape-expression*:

- The *escape-expression* must be a string of length 1.[48]
- The *pattern-expression* must not contain the escape character except when followed by the escape character, percent, or underscore.

  For example, if '+' is the escape character, any occurrences of '+' other than '++', '+_', or '+%' in the *pattern-expression* is an error.
- The *escape-expression* can be a parameter marker.

The following example shows the effect of successive occurrences of the escape character, which in this case is the plus sign (+).

Table 29. Effect of successive occurrences of the escape character.

When the pattern string is...	The actual pattern is...
+%	A percent sign
++%	A plus sign followed by zero or more arbitrary characters
+++%	A plus sign followed by a percent sign

Examples

Example 1

48. If it is NUL-terminated, a C character string variable of length 2 can be specified.

Search for the string 'SYSTEMS' appearing anywhere within the PROJNAME column in the PROJECT table.

```
SELECT PROJNAME
  FROM PROJECT
  WHERE PROJECT.PROJNAME LIKE '%SYSTEMS%'
```

Example 2

Search for a string with a first character of 'J' that is exactly two characters long in the FIRSTNME column of the EMPLOYEE table.

```
SELECT FIRSTNME
     FROM EMPLOYEE
     WHERE EMPLOYEE.FIRSTNME LIKE 'J_'
```

Example 3

In this example:

```
SELECT *
     FROM TABLEY
     WHERE C1 LIKE 'AAAA+%BBB%' ESCAPE '+'
```

'+' is the escape character and indicates that the search is for a string that starts with 'AAAA%BBB'. The '+%' is interpreted as a single occurrence of '%' in the pattern.

Example 4

In the following table of EBCDIC examples, assume COL1 is mixed data. The table shows the results when the predicates in the first column are evaluated using the COL1 values from the second column:

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaa SO AB%C SI'	'aaa SO ABDZC SI'	True
WHERE COL1 LIKE 'aaa SO AB SI % SO C SI'	'aaa SO AB SI dzx SO C SI'	True
WHERE COL1 LIKE 'a% SO C SI'	'a SO C SI'	True
	'ax SO C SI'	True
	'ab SO DE SI fg SO C SI'	True
WHERE COL1 LIKE 'a _ SO C SI'	'a% SO C SI'	True
	'a SO X C SI'	False
WHERE COL1 LIKE 'a SO __ C SI'	'a SO X C SI'	True
	'ax SO C SI'	False
WHERE COL1 LIKE 'SO SI'	Empty string	True
WHERE COL1 LIKE 'ab SO C SI _'	'ab SO C SI d'	True
	'ab SO SI SO C SI d'	True

RV3F001-0

Example 5

Assume that a distinct type named ZIP_TYPE with a source data type of CHAR(5) exists and an ADDRZIP column with data type ZIP_TYPE exists in some table TABLEY. The following statement selects the row if the zip code (ADDRZIP) begins with '9555'.

```
SELECT *
  FROM TABLEY
  WHERE CHAR(ADDRZIP) LIKE '9555%'
```

Example 6

The RESUME column in sample table EMP_RESUME is defined as a CLOB. If the variable LASTNAME has a value of 'JONES', the following statement selects the RESUME column when the string JONES appears anywhere in the column.

```
SELECT RESUME
  FROM EMP_RESUME
  WHERE RESUME LIKE '%'||LASTNAME||'%'
```

NULL predicate

The NULL predicate tests for null values.

The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false.

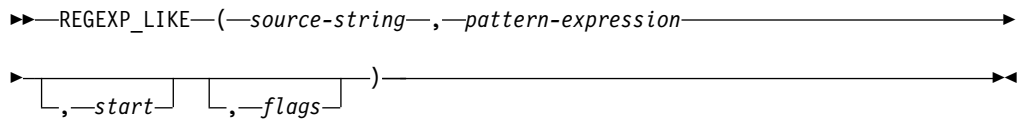
If NOT is specified, the result is reversed.

Examples

```
EMPLOYEE.PHONE IS NULL

SALARY IS NOT NULL
```

REGEXP_LIKE predicate

The REGEXP_LIKE predicate searches for a regular expression pattern in a string.



If the *pattern-expression* is found, the result is true. If the *pattern-expression* is not found, the result is false. If *source-string* and *pattern-expression* are empty strings, the result is true. If *source-string* or *pattern-expression* is the empty string (but not both), the result is false. If the value of any of the arguments is null, the result is unknown.

source-string
An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of a string must not be greater than 1 gigabyte.

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of the string must not be greater than 32K.

A valid *pattern-expression* consists of a set of characters and control characters that describe the pattern of the search. For a description of the valid control characters, see “Regular expression control characters” on page 220.

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before searching for the regular expression pattern. For more information about converting to INTEGER, see “INTEGER or INT” on page 384. The value of the integer must be greater than or equal to 1. If the value of the integer is greater than the actual length of the *source-string*, the result of the predicate is false.

flags
An expression that specifies flags that control aspects of the pattern matching. The expression must return a value that is a built-in character string or graphic string data type. A character string with the FOR BIT DATA attribute or a binary string is not supported. The string can include one or more valid flag values and the combination of flag values must be valid. An empty string is the same as the value 'c'.

For a description of the valid flag characters, see “Regular expression flag values” on page 220.

Regular expression flag values

The following table describes the supported flag values.

Table 30. Flag Values

Flag value	Description
c	Specifies that matching is case sensitive. This is the default value if neither 'c' nor 'i' is specified. This value must not be specified with a value of 'i'.
i	Specifies that matching is case insensitive. This value must not be specified with a value of 'c'.
m	Specifies that the input data could contain more than one line. By default, the '^' and the '$' in a pattern will only match the start and the end, respectively, of the input string. If this flag is set, "^" and "$" will also match at the start and end of each line within the input string.
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' in a pattern will not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single line terminator, and will match a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

Regular expression control characters

The regular expression processing is performed using the International Components for Unicode (ICU) regular expression API on Unicode data with regular expression patterns that use the control characters listed below.

Note that only halfwidth characters are recognized. Any fullwidth characters that correspond to the characters in these tables will not be recognized.

Table 31. Regular Expression Metacharacters

Character	Outside of sets	[Inside sets]	Description
\a	✓	✓	Match a BELL, \u0007
\A	✓		Match at the beginning of the input. Differs from ^ in that \A will not match after a new line within the input.
\b	✓		Match if the current position is a word boundary. Boundaries occur at the transitions between word (\w) and non-word (\W) characters, with combining marks ignored.
\B	✓		Match if the current position is not a word boundary.
\cX	✓	✓	Match a control-X character.
\d	✓	✓	Match any character with the Unicode General Category of Nd (Number, Decimal Digit.)
\D	✓	✓	Match any character that is not a decimal digit.
\e	✓	✓	Match an ESCAPE, \u001B.
\E	✓	✓	Terminates a \Q ... \E quoted sequence.
\f	✓	✓	Match a FORM FEED, \u000C.
\G	✓		Match if the current position is at the end of the previous match.
\n	✓	✓	Match a LINE FEED, \u000A.
\N{UNICODE CHARACTER NAME}	✓	✓	Match the named character.

Table 31. Regular Expression Metacharacters (continued)

Character	Outside of sets	[Inside sets]	Description
\p{UNICODE PROPERTY NAME}	✓	✓	Match any character with the specified Unicode Property.
\P{UNICODE PROPERTY NAME}	✓	✓	Match any character not having the specified Unicode Property.
\Q	✓	✓	Quotes all following characters until \E.
\r	✓	✓	Match a CARRIAGE RETURN, \u000D
\s	✓	✓	Match a white space character. White space is defined as [\t\n\f\r\p{Z}].
\S	✓	✓	Match a non-white space character.
\t	✓	✓	Match a HORIZONTAL TABULATION, \u0009.
\uhhhh	✓	✓	Match the character with the hex value hhhh.
\Uhhhhhhhh	✓	✓	Match the character with the hex value hhhhhhhh. Exactly eight hex digits must be provided, even though the largest Unicode code point is \U0010ffff.
\w	✓	✓	Match a word character. Word characters are [\p{Alphabetic}\p{Mark}\p{Decimal_Number} \p{Connector_Punctuation}\u200c\u200d].
\W	✓	✓	Match a non-word character.
\x{hhhh}	✓	✓	Match the character with hex value hhhh. From one to six hex digits may be supplied.
\xhh	✓	✓	Match the character with two digit hex value hh
\X	✓		Match a Grapheme Cluster
\Z	✓		Match if the current position is at the end of input, but before the final line terminator, if one exists.
\z	✓		Match if the current position is at the end of input.
\n	✓		Back Reference. Match whatever the nth capturing group matched. n must be a number > 1 and < total number of capture groups in the pattern.
\0ooo	✓	✓	Match an Octal character. 'ooo' is from one to three octal digits. 0377 is the largest allowed Octal character. The leading zero is required; it distinguishes Octal constants from back references.
[pattern]	✓	✓	Match any one character from the set.
.	✓		Match any character.
^	✓		Match at the beginning of a line.
$	✓		Match at the end of a line.
\	✓		Quotes the following character. Characters that must be quoted to be treated as literals are * ? + [( ) { } ^ $ | \ .
\		✓	Quotes the following character. Characters that must be quoted to be treated as literals are [ ] \ Characters that may need to be quoted, depending on the context are - &

Table 32. Regular Expression Operators

Operator	Description
|	Alternation. A|B matches either A or B.
*	Match 0 or more times. Match as many times as possible.
+	Match 1 or more times. Match as many times as possible.
?	Match zero or one times. Prefer one
{n}	Match exactly n times
{n,}	Match at least n times. Match as many times as possible.
{n,m}	Match between n and m times. Match as many times as possible, but not more than m.
*?	Match 0 or more times. Match as few times as possible.
+?	Match 1 or more times. Match as few times as possible.
??	Match zero or one times. Prefer zero.
{n}?	Match exactly n times
{n,}?	Match at least n times, but no more than required for an overall pattern match
{n,m}?	Match between n and m times. Match as few times as possible, but not less than n.
*+	Match 0 or more times. Match as many times as possible when first encountered, do not retry with fewer even if overall match fails (Possessive Match)
++	Match 1 or more times. Possessive match
?+	Match zero or one times. Possessive match
{n}+	Match exactly n times
{n,}+	Match at least n times. Possessive Match.
{n,m}+	Match between n and m times. Possessive Match.
(...)	Capturing parentheses. Range of input that matched the parenthesized subexpression is available after the match.
(?: ...)	Non-capturing parentheses. Groups the included pattern, but does not provide capturing of matching text. Somewhat more efficient than capturing parentheses.
(?> ...)	Atomic-match parentheses. First match of the parenthesized subexpression is the only one tried; if it does not lead to an overall pattern match, back up the search for a match to a position before the "(?>"
(?# ...)	Free-format comment (?# comment).
(?= ...)	Look-ahead assertion. True if the parenthesized pattern matches at the current input position, but does not advance the input position.
(?! ...)	Negative look-ahead assertion. True if the parenthesized pattern does not match at the current input position. Does not advance the input position.
(?<= ...)	Look-behind assertion. True if the parenthesized pattern matches text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the look-behind pattern must not be unbounded (no * or + operators.)
(?<! ...)	Negative Look-behind assertion. True if the parenthesized pattern does not match text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the look-behind pattern must not be unbounded (no * or + operators.)
(?ismwx-ismwx: ...)	Flag settings. Evaluate the parenthesized expression with the specified flags enabled or disabled.
(?ismwx-ismwx)	Flag settings. Change the flag settings. Changes apply to the portion of the pattern following the setting. For example, (?i) changes to a case insensitive match.

Table 33. Set Expressions (Character Classes)

Example	Description
[abc]	Match any of the characters a, b or c
[^abc]	Negation - match any character except a, b or c
[A-M]	Range - match any character from A to M. The characters to include are determined by Unicode code point ordering.
[\u0000-\U0010ffff]	Range - match all characters.
[\p{Letter}] [\p{General_Category=Letter}] [\p{L}]	Characters with Unicode Category = Letter. All forms shown are equivalent.
[\P{Letter}]	Negated property. (Upper case \P) Match everything except Letters.
[\p{numeric_value=9}]	Match all numbers with a numeric value of 9. Any Unicode Property may be used in set expressions.
[\p{Letter}&&\p{script=cyrillic}]	Logical AND or intersection. Match the set of all Cyrillic letters.
[\p{Letter}--\p{script=latin}]	Subtraction. Match all non-Latin letters.
[[a-z][A-Z][0-9]] [[a-zA-Z0-9]]	Implicit Logical OR or Union of Sets. The examples match ASCII letters and digits. The two forms are equivalent.
[:script=Greek:]	Alternate POSIX-like syntax for properties. Equivalent to \p{script=Greek}

Notes

Prerequisites: In order to use the REGEXP_LIKE predicate, the International Components for Unicode (ICU) option must be installed

Processing: The regular expression processing is done using the International Components for Unicode (ICU) regular expression interface. For more information see, http://userguide.icu-project.org/strings/regexp.

If only three arguments are specified, the third argument may be a *start* or *flags* argument. If the third argument is a string, it is interpreted as a *flags* argument. Otherwise, it is interpreted as a *start* argument.

Examples

- *Example 1*: Select the employee number where the last name is spelled LUCCHESSI, LUCHESSI, or LUCHESI from the EMPLOYEE table without considering upper or lower case letters.

```
SELECT EMPNO FROM EMPLOYEE
 WHERE REGEXP_LIKE(LASTNAME,'luc+?hes+?i','i')
```

  The result is 1 row with EMPNO value '000110'.

- *Example 2*: Select any invalid product identifier values from the PRODUCT table. The expected format is 'nnn-nnn-nn' where 'n' is a digit from 0 to 9.

```
SELECT PID FROM PRODUCT
 WHERE NOT REGEXP_LIKE(pid,'[0-9]{3}-[0-9]{3}-[0-9]{2}')
```

  The result is 0 rows because all the product identifiers match the pattern.

Trigger event predicates

A trigger event predicate is used in a triggered action to test the event that activated the trigger. It is only allowed in the triggered action of a CREATE TRIGGER statement.

DELETING
True if the trigger was activated by a delete operation. False otherwise.

INSERTING
True if the trigger was activated by an insert operation. False otherwise.

UPDATING
True if the trigger was activated by an update operation. False otherwise.

Notes

A trigger event predicate can be used anywhere in the triggered action of a CREATE TRIGGER statement.

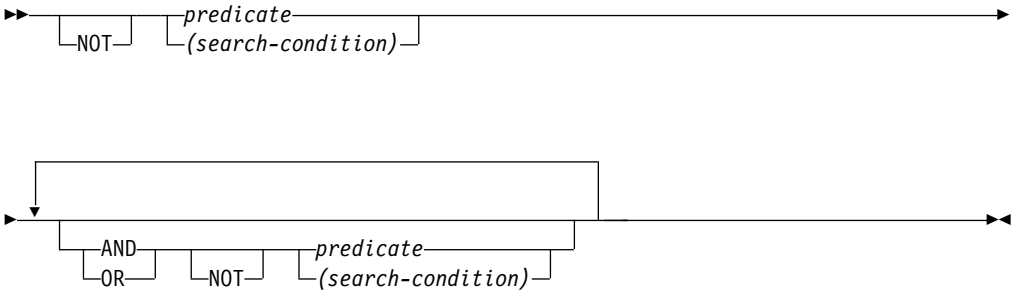
Example

The following trigger, by using trigger event predicates in the routine body, increments the number of employees each time a new person is hired (that is, each time a new row is inserted into the EMPLOYEE table), decrements the number of employees each time an employee leaves the company, and raises an error when an update occurs that would result in a salary increase greater than ten percent of the current salary.

```
CREATE TRIGGER HIRED
 AFTER INSERT OR DELETE OR UPDATE OF SALARY ON EMPLOYEE
 REFERENCING NEW AS N OLD AS O FOR EACH ROW
   BEGIN
     IF INSERTING
       THEN UPDATE COMPANY_STATS SET NBREMP = NBREMP + 1;
     END IF;
     IF DELETING
       THEN UPDATE COMPANY_STATS SET NBREMP = NBREMP - 1;
     END IF;
     IF UPDATING AND (N.SALARY > 1.1 * O.SALARY)
        THEN SIGNAL SQLSTATE '75000' SET MESSAGE_TEXT = 'Salary increase > 10%'
     END IF;
   END
```

Search conditions

A *search condition* specifies a condition that is true, false, or unknown about a given row or group.



The result of a search condition is derived by application of the specified *logical operators* (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in the following table in which P and Q are any predicates:

Table 34. Truth Tables for AND and OR

P	Q	P AND Q	P OR Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

Examples

In the examples, the numbers on the second line indicate the order in which the operators are evaluated.

Example 1

```
MAJPROJ = 'MA2100' AND DEPTNO = 'D11'  OR  DEPTNO = 'B03' OR DEPTNO = 'E11'
                      ↑                    ↑                 ↑
                      1                  2 or 3            2 or 3
```

Example 2

```
MAJPROJ = 'MA2100'  AND  (DEPTNO = 'D11'  OR  DEPTNO = 'B03')  OR  DEPTNO = 'E11'
                    ↑                    ↑                    ↑
                    2                    1                    3
```

Chapter 3. Built-in functions

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the *built-in functions* listed in the following tables.

For more information about functions, see "Functions" on page 158.

Table 35. Aggregate Functions

Function	Description	Reference
ARRAY_AGG	Aggregates a set of elements into an array.	"ARRAY_AGG" on page 238
AVG	Returns the average of a set of numbers	"AVG" on page 240
COUNT	Returns the number of rows or values in a set of rows or values	"COUNT" on page 242
COUNT_BIG	Returns the number of rows or values in a set of rows or values (COUNT_BIG is similar to COUNT except that the result can be greater than the maximum value of integer)	"COUNT_BIG" on page 243
GROUPING	Returns a value that indicates whether a row was generated as a summary row for a grouping set.	"GROUPING" on page 244
MAX	Returns the maximum value in a set of values in a group	"MAX" on page 246
MIN	Returns the minimum value in a set of values in a group	"MIN" on page 247
STDDEV	Returns the biased standard deviation of a set of numbers	"STDDEV_POP or STDDEV" on page 249
STDDEV_SAMP	Returns the sample standard deviation of a set of numbers	"STDDEV_SAMP" on page 250
SUM	Returns the sum of a set of numbers	"SUM" on page 251
VARIANCE or VAR	Returns the biased variance of a set of numbers	"VAR_POP or VARIANCE or VAR" on page 252
VARIANCE_SAMP or VAR_SAMP	Returns the sample variance of a set of numbers	"VARIANCE_SAMP or VAR_SAMP" on page 253
XMLAGG	Returns an XML sequence containing an item for each non-null value in a set of XML values.	"XMLAGG" on page 254
XMLGROUP	Returns an XML value that is a well-formed XML document.	"XMLGROUP" on page 256

Table 36. Cast Scalar Functions

Function	Description	Reference
BIGINT	Returns a big integer representation of a number	"BIGINT" on page 269
BINARY	Returns a BINARY representation of a string of any type	"BINARY" on page 271
BLOB	Returns a BLOB representation of a string of any type	"BLOB" on page 275
CHAR	Returns a CHARACTER representation of a value	"CHAR" on page 279
CLOB	Returns a CLOB representation of a value	"CLOB" on page 287
DATE	Returns a DATE from a value	"DATE" on page 307

Table 36. Cast Scalar Functions (continued)

Function	Description	Reference
DBCLOB	Returns a DBCLOB representation of a string	"DBCLOB" on page 316
DECFLOAT	Returns a DECFLOAT representation of a number	"DECFLOAT" on page 323
DECIMAL	Returns a DECIMAL representation of a number	"DECIMAL or DEC" on page 326
DOUBLE_PRECISION or DOUBLE	Returns a DOUBLE PRECISION representation of a number	"DOUBLE_PRECISION or DOUBLE" on page 344
FLOAT	Returns a FLOAT representation of a number	"FLOAT" on page 358
GRAPHIC	Returns a GRAPHIC representation of a string	"GRAPHIC" on page 367
INTEGER or INT	Returns an INTEGER representation of a number	"INTEGER or INT" on page 384
REAL	Returns a REAL representation of a number	"REAL" on page 453
ROWID	Returns a Row ID from a value	"ROWID" on page 478
SMALLINT	Returns a SMALLINT representation of a number	"SMALLINT" on page 491
TIME	Returns a TIME from a value	"TIME" on page 504
TIMESTAMP	Returns a TIMESTAMP from a value or a pair of values	"TIMESTAMP" on page 505
TIMESTAMP_ISO	Returns a timestamp value from a datetime value	"TIMESTAMP_ISO" on page 512
VARBINARY	Returns a VARBINARY representation of a string of any type	"VARBINARY" on page 528
VARCHAR	Returns a VARCHAR representative of a value	"VARCHAR" on page 531
VARGRAPHIC	Returns a VARGRAPHIC representation of a value	"VARGRAPHIC" on page 547
ZONED	Returns a zoned decimal representation of a number	"ZONED" on page 589

Table 37. Datalink Scalar Functions

Function	Description	Reference
DLCOMMENT	Returns the comment value from a DataLink value	"DLCOMMENT" on page 335
DLLINKTYPE	Returns the link type value from a DataLink value	"DLLINKTYPE" on page 336
DLURLCOMPLETE	Returns the complete URL value from a DataLink value with a link type of URL	"DLURLCOMPLETE" on page 337
DLURLPATH	Returns the path and file name necessary to access a file within a given server from a DataLink value with a linktype of URL	"DLURLPATH" on page 338
DLURLPATHONLY	Returns the path and file name necessary to access a file within a given server from a DataLink value with a linktype of URL without a file access token	"DLURLPATHONLY" on page 339
DLURLSCHEME	Returns the scheme from a DataLink value with a linktype of URL	"DLURLSCHEME" on page 340
DLURLSERVER	Returns the file server from a DataLink value with a linktype of URL	"DLURLSERVER" on page 341
DLVALUE	Returns a DataLink value	"DLVALUE" on page 342

Table 38. Datetime Scalar Functions

Function	Description	Reference
ADD_MONTHS	Returns a date that represents the date argument plus the number of months argument	"ADD_MONTHS" on page 262
CURDATE	Returns a date based on a reading of the time-of-day clock	"CURDATE" on page 302
CURTIME	Returns a time based on a reading of the time-of-day clock	"CURTIME" on page 303
DAY	Returns the day part of a value	"DAY" on page 309
DAYNAME	Returns the name of the day part of a value	"DAYNAME" on page 310
DAYOFMONTH	Returns an integer that represents the day of the month	"DAYOFMONTH" on page 311
DAYOFWEEK	Returns the day of the week from a value, where 1 is Sunday and 7 is Saturday	"DAYOFWEEK" on page 312
DAYOFWEEK_ISO	Returns the day of the week from a value, where 1 is Monday and 7 is Sunday	"DAYOFWEEK_ISO" on page 313
DAYOFYEAR	Returns the day of the year from a value	"DAYOFYEAR" on page 314
DAYS	Returns an integer representation of a date	"DAYS" on page 315
EXTRACT	Returns a specified portion of a datetime value	"EXTRACT" on page 356
HOUR	Returns the hour part of a value	"HOUR" on page 376
JULIAN_DAY	Returns an integer value representing a number of days from January 1, 4712 B.C. to the date specified in the argument	"JULIAN_DAY" on page 386
LAST_DAY	Returns a date or timestamp that represents the last day of the month of the argument	"LAST_DAY" on page 388
MICROSECOND	Returns the microsecond part of a value	"MICROSECOND" on page 410
MIDNIGHT_SECONDS	Returns an integer value representing the number of seconds between midnight and a specified time value	"MIDNIGHT_SECONDS" on page 411
MINUTE	Returns the minute part of a value	"MINUTE" on page 413
MONTH	Returns the month part of a value	"MONTH" on page 416
MONTHNAME	Returns the name of the month part of a value	"MONTHNAME" on page 417
MONTHS_BETWEEN	Returns an estimate of the number of months between two dates	"MONTHS_BETWEEN" on page 418
NEXT_DAY	Returns a date or timestamp value that represents the first weekday after the day named by the second argument	"NEXT_DAY" on page 432
NOW	Returns a timestamp based on a reading of the time-of-day clock	"NOW" on page 435
QUARTER	Returns an integer that represents the quarter of the year in which a date resides	"QUARTER" on page 449
ROUND_TIMESTAMP	Returns a timestamp rounded to the specified unit.	"ROUND_TIMESTAMP" on page 475
SECOND	Returns the seconds part of a value	"SECOND" on page 487
TIMESTAMP_FORMAT	Returns a timestamp from a character string representation of a timestamp according to the specified format of the string.	"TIMESTAMP_FORMAT" on page 507

Table 38. Datetime Scalar Functions (continued)

Function	Description	Reference
TIMESTAMPDIFF	Returns an estimated number of intervals based on the difference between two timestamps	"TIMESTAMPDIFF" on page 513
TRUNC_TIMESTAMP	Returns a timestamp truncated to the specified unit.	"TRUNC_TIMESTAMP" on page 524
VARCHAR_FORMAT	Returns a character string representation of a timestamp, with the string in a specified format	"VARCHAR_FORMAT" on page 537
WEEK	Returns the week of the year from a value, where the week starts with Sunday	"WEEK" on page 552
WEEK_ISO	Returns the week of the year from a value, where the week starts with Monday	"WEEK_ISO" on page 553
YEAR	Returns the year part of a value	"YEAR" on page 588

Table 39. Partitioning Scalar Functions

Function	Description	Reference
DATAPARTITIONNAME	Returns the partition name where a row is located	"DATAPARTITIONNAME" on page 305
DATAPARTITIONNUM	Returns the partition number of a row	"DATAPARTITIONNUM" on page 306
DBPARTITIONNAME	Returns the relational database name where a row is located	"DBPARTITIONNAME" on page 321
DBPARTITIONNUM	Returns the node number of a row	"DBPARTITIONNUM" on page 322
HASH	Returns the partition number of a set of values	"HASH" on page 372
HASHED_VALUE	Returns the partition map index number of a row	"HASHED_VALUE" on page 373

Table 40. Miscellaneous Scalar Functions

Function	Description	Reference
CARDINALITY	Returns a value representing the number of elements of an array	"CARDINALITY" on page 277
COALESCE	Returns the first argument that is not null	"COALESCE" on page 292
CONTAINS	Returns an indication of whether a match was found in a text index.	"CONTAINS" on page 296
DATABASE	Returns the current server	"DATABASE" on page 304
GENERATE_UNIQUE	Returns a bit character string that is unique compared to any other execution of the function	"ATAN2" on page 268
GET_BLOB_FROM_FILE	Returns a BLOB locator containing the data from a source stream file or a source physical file member.	"GET_BLOB_FROM_FILE" on page 362
GET_CLOB_FROM_FILE	Returns a CLOB locator containing the data from a source stream file or a source physical file member.	"GET_CLOB_FROM_FILE" on page 363
GET_DBCLOB_FROM_FILE	Returns a DBCLOB locator containing the data from a source stream file or a source physical file member.	"GET_DBCLOB_FROM_FILE" on page 364

Table 40. Miscellaneous Scalar Functions (continued)

Function	Description	Reference
GET_XML_FILE	Returns a BLOB locator containing the data from a source stream file or a source physical file member with the data converted to UTF-8.	"GET_XML_FILE" on page 365
HEX	Returns a hexadecimal representation of a value	"HEX" on page 374
IDENTITY_VAL_LOCAL	Returns the most recently assigned value for an identity column	"IDENTITY_VAL_LOCAL" on page 377
IFNULL	Returns the first argument that is not null	"IFNULL" on page 381
LENGTH	Returns the length of a value	"LENGTH" on page 392
MAX	Returns the maximum value in a set of values	"MAX" on page 408
MAX_CARDINALITY	Returns a value representing the maximum number of elements on array can contain.	"MAX_CARDINALITY" on page 409
MIN	Returns the minimum value in a set of values	"MIN" on page 412
NULLIF	Returns a null value if the arguments are equal, otherwise it returns the value of the first argument	"NULLIF" on page 436
RAISE_ERROR	Raises an error with the specified SQLSTATE and message text	"RAISE_ERROR" on page 451
RID	Returns the relative record number of a row as BIGINT.	"RID" on page 470
RRN	Returns the relative record number of a row as DECIMAL(15,0)	"RRN" on page 482
SCORE	Returns an indication of how frequently a match was found in a text index.	"SCORE" on page 484
TRIM_ARRAY	Returns a copy of an array from which a number of elements have been removed from the end of the array.	"TRIM_ARRAY" on page 521
VALUE	Returns the first argument that is not null	"VALUE" on page 527

Table 41. MQSeries Scalar Functions

Function	Description	Reference
MQREAD	Returns a message from a specified MQSeries® location (return value of VARCHAR) without removing the message from the queue	"MQREAD" on page 420
MQREADCLOB	Returns a message from a specified MQSeries location (return value of CLOB) without removing the message from the queue	"MQREADCLOB" on page 422
MQRECEIVE	Returns a message from a specified MQSeries location (return value of VARCHAR) with removal of message from the queue	"MQRECEIVE" on page 424
MQRECEIVECLOB	Returns a message from a specified MQSeries location (return value of CLOB) with removal of message from the queue	"MQRECEIVECLOB" on page 426
MQSEND	Sends a message to a specified MQSeries location	"MQSEND" on page 428

Table 42. Numeric Scalar Functions

Function	Description	Reference
ABS	Returns the absolute value of a number	"ABS" on page 260
ACOS	Returns the arc cosine of a number, in radians	"ACOS" on page 261
ANTILOG	Returns the anti-logarithm (base 10) of a number	"ANTILOG" on page 263
ASIN	Returns the arc sine of a number, in radians	"ASIN" on page 265
ATAN	Returns the arc tangent of a number, in radians	"ATAN" on page 266
ATANH	Returns the hyperbolic arc tangent of a number, in radians	"ATANH" on page 267
ATAN2	Returns the arc tangent of x and y coordinates as an angle expressed in radians	"ATAN2" on page 268
BITAND, BITANDNOT, BITOR, BITXOR, BITNOT	Return a base 10 value based on a bitwise operation using the two's complement representation of the input arguments	"BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT" on page 272
CEILING	Returns the smallest integer value that is greater than or equal to a number	"CEILING" on page 278
COMPARE_DECFLOAT	Returns an indication of how two decimal floating-point values compare	"COMPARE_DECFLOAT" on page 293
COS	Returns the cosine of a number	"COS" on page 299
COSH	Returns the hyperbolic cosine of a number	"COSH" on page 300
COT	Returns the cotangent of a number	"COT" on page 301
DECFLOAT_SORTKEY	Returns an integer value that can be used to sort decimal floating-point values.	"DECFLOAT_SORTKEY" on page 325
DEGREES	Returns the number of degrees of an angle	"DEGREES" on page 332
DIGITS	Returns a character-string representation of the absolute value of a number	"DIGITS" on page 334
EXP	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument	"EXP" on page 355
FLOOR	Returns the largest integer value that is less than or equal to a number	"FLOOR" on page 359
LN	Returns the natural logarithm of a number	"LN" on page 394
LOG10	Returns the common logarithm (base 10) of a number	"LOG10" on page 401
MOD	Returns the remainder of the first argument divided by the second argument	"MOD" on page 414
MULTIPLY_ALT	Multiplies the first argument by the second argument and returns the product	"MULTIPLY_ALT" on page 430
NORMALIZE_DECFLOAT	Returns a decimal floating-point value in its simplest form	"NORMALIZE_DECFLOAT" on page 434
PI	Returns the value of π	"PI" on page 441
POWER	Returns the result of raising the first argument to the power of the second argument	"POWER" on page 446
QUANTIZE	Returns a decimal floating-point value formatted according to a provided value	"QUANTIZE" on page 447
RADIANS	Returns the number of radians for an argument that is expressed in degrees	"RADIANS" on page 450

Table 42. Numeric Scalar Functions (continued)

Function	Description	Reference
RAND	Returns a random number	"RAND" on page 452
ROUND	Returns a numeric value that has been rounded to the specified number of decimal places	"ROUND" on page 473
SIGN	Returns the sign of a number	"SIGN" on page 488
SIN	Returns the sine of a number	"SIN" on page 489
SINH	Returns the hyperbolic sine of a number	"SINH" on page 490
SQRT	Returns the square root of a number	"SQRT" on page 495
TAN	Returns the tangent of a number	"TAN" on page 502
TANH	Returns the hyperbolic tangent of a number	"TANH" on page 503
TOTALORDER	Returns an ordering indication for two decimal floating-point values	"TOTALORDER" on page 516
TRUNCATE or TRUNC	Returns a number value that has been truncated at a specified number of decimal places	"TRUNCATE or TRUNC" on page 522

Table 43. String Scalar Functions

Function	Description	Reference
ASCII	Returns the ASCII code value of the leftmost character of the argument as an integer	"ASCII" on page 264
BIT_LENGTH	Returns the length of a string expression in bits	"BIT_LENGTH" on page 274
CHARACTER_LENGTH	Returns the length of a string expression	"CHARACTER_LENGTH" on page 285
CHR	Returns the character that has the ASCII code value specified by the argument.	"CHR" on page 286
CONCAT	Returns a string that is the concatenation of two strings	"CONCAT" on page 295
DECRYPT_BIT, DECRYPT_BINARY, DECRYPT_CHAR, and DECRYPT_DB	Decrypts an encrypted string	"DECRYPT_BIT, DECRYPT_BINARY, DECRYPT_CHAR and DECRYPT_DB" on page 329
DIFFERENCE	Returns a value representing the difference between the sounds of two strings	"DIFFERENCE" on page 333
ENCRYPT and ENCRYPT_RC2	Encrypts a string using the RC2 encryption algorithm	"ENCRYPT_RC2" on page 349
ENCRYPT_AES	Encrypts a string using the AES encryption algorithm	"ENCRYPT_AES" on page 346
ENCRYPT_TDES	Encrypts a string using the Triple DES encryption algorithm	"ENCRYPT_TDES" on page 352
GETHINT	Returns a hint from an encrypted string	"GETHINT" on page 366
INSERT	Returns a string where a substring is deleted and a new string inserted in its place	"INSERT" on page 382
LAND	Returns a string that is the logical AND of the argument strings	"LAND" on page 387
LCASE	Returns a string in which all the characters have been converted to lowercase characters	"LCASE" on page 389
LEFT	Returns the leftmost characters from the string	"LEFT" on page 390

Table 43. String Scalar Functions (continued)

Function	Description	Reference
LNOT	Returns a string that is the logical NOT of the argument string	"LNOT" on page 395
LOCATE	Returns the starting position of one string within another string	"LOCATE" on page 396
LOCATE_IN_STRING	The LOCATE_IN_STRING function returns the starting position of a string within another string.	"LOCATE_IN_STRING" on page 398
LOR	Returns a string that is the logical OR of the argument strings	"LOR" on page 402
LOWER	Returns a string in which all the characters have been converted to lowercase characters	"LOWER" on page 403
LTRIM	Returns a string in which blanks or hexadecimal zeroes have been removed from the beginning of another string	"LTRIM" on page 407
OCTET_LENGTH	Returns the length of a string expression in octets	"OCTET_LENGTH" on page 437
OVERLAY	Returns a string where length characters have been deleted from source-string beginning at start and where insert-string has been inserted into source-string beginning at start.	"OVERLAY" on page 438
POSITION	Returns the starting position of one string within another string	"POSITION" on page 442
POSSTR	Returns the starting position of one string within another string	"POSSTR" on page 444
REGEXP_COUNT	Returns a count of the number of times that a regular expression pattern is matched in a string	"REGEXP_COUNT" on page 455
REGEXP_INSTR	Returns the starting position or the position after the end of the matched substring	"REGEXP_INSTR" on page 457
REGEXP_REPLACE	Returns a string where occurrences of the regular expression pattern found in the string are replaced by another string	"REGEXP_REPLACE" on page 460
REGEXP_SUBSTR	Returns one occurrence of a substring of a string that matches the regular expression pattern	"REGEXP_SUBSTR" on page 463
REPEAT	Returns a string composed of another string repeated a number of times	"REPEAT" on page 466
REPLACE	Returns a string where all occurrences of one string are replaced by another string	"REPLACE" on page 468
RIGHT	Returns the rightmost characters from the string	"RIGHT" on page 471
RTRIM	Returns a string in which blanks or hexadecimal zeroes have been removed from the end of another string	"RTRIM" on page 483
SOUNDEX	Returns a character code representing the sound of the words in the argument	"SOUNDEX" on page 493
SPACE	Returns a character string that consists of a specified number of blanks	"SPACE" on page 494
STRIP	Removes blanks or another specified character from the end or beginning of a string expression	"STRIP" on page 496
SUBSTR	Returns a substring of a string	"SUBSTR" on page 497
SUBSTRING	Returns a substring of a string	"SUBSTRING" on page 500

Table 43. String Scalar Functions (continued)

Function	Description	Reference
TRANSLATE	Returns a string in which one or more characters in a string are converted to other characters	"TRANSLATE" on page 517
TRIM	Removes blanks or another specified character from the beginning, the end, or both the beginning and the end of a string expression	"TRIM" on page 519
UCASE	Returns a string in which all the characters have been converted to uppercase characters	"UCASE" on page 525
UPPER	Returns a string in which all the characters have been converted to uppercase characters	"UPPER" on page 526
VARBINARY_FORMAT	Returns a binary string representation of a character string that has been formatted using a *format-string*	"VARBINARY_FORMAT" on page 529
VARCHAR_FORMAT_BINARY	Returns a character string representation of a bit string that has been formatted using a *format-string*	"VARCHAR_FORMAT_BINARY" on page 546
XOR	Returns a string that is the logical XOR of the argument strings	"XOR" on page 583

Table 44. XML Scalar Functions

Function	Description	Reference
XMLATTRIBUTES	Constructs XML attributes from the arguments.	"XMLATTRIBUTES" on page 556
XMLCOMMENT	Returns an XML value with the input argument as the content.	"XMLCOMMENT" on page 557
XMLCONCAT	Returns an XML sequence containing the concatenation of a variable number of XML input arguments.	"XMLCONCAT" on page 558
XMLDOCUMENT	Returns an XML value that is a well-formed XML document.	"XMLDOCUMENT" on page 560
XMLELEMENT	Returns an XML value that is an XML element.	"XMLELEMENT" on page 561
XMLFOREST	Returns an XML value that is a sequence of XML elements.	"XMLFOREST" on page 565
XMLNAMESPACES	Constructs namespace declarations from the arguments.	"XMLNAMESPACES" on page 568
XMLPARSE	Returns an XML value by parsing the arguments as an XML document.	"XMLPARSE" on page 570
XMLPI	Returns an XML value with a single processing instruction.	"XMLPI" on page 572
XMLROW	Returns an XML value that is a well-formed XML document.	"XMLROW" on page 573
XMLSERIALIZE	Returns a serialized XML value as the specified data type.	"XMLSERIALIZE" on page 575
XMLTEXT	Returns an XML value that contains the value of the input argument.	"XMLTEXT" on page 578
XMLVALIDATE	Returns an XML value that has been augmented with information obtained from XML schema validation.	"XMLVALIDATE" on page 579

Table 44. XML Scalar Functions (continued)

Function	Description	Reference
XSLTRANSFORM	Transforms an XML document into a different data format.	"XSLTRANSFORM" on page 584

Table 45. Table Functions

Function	Description	Reference
BASE_TABLE	Returns a table with one row containing the table name and schema name for the specified alias name.	"BASE_TABLE" on page 592
MQREADALL	Returns a table containing the messages and message metadata from a specified MQSeries location with a VARCHAR column and without removing the messages from the queue	"MQREADALL" on page 594
MQREADALLCLOB	Returns a table containing the messages and message metadata from a specified MQSeries location with a CLOB column and without removing the messages from the queue	"MQREADALLCLOB" on page 596
MQRECEIVEALL	Returns a table containing the messages and message metadata from a specified MQSeries location with a VARCHAR column and with removal of messages from the queue	"MQRECEIVEALL" on page 598
MQRECEIVEALLCLOB	Returns a table containing the messages and message metadata from a specified MQSeries location with a CLOB column and with removal of messages from the queue	"MQRECEIVEALLCLOB" on page 601
XMLTABLE	Returns a table from the evaluation of an XPath expression	"XMLTABLE" on page 604

Aggregate functions

An aggregate function takes a set of values (like a column of data) and returns a single value result from the set of values.

The following information applies to all aggregate functions other than COUNT(*) and COUNT_BIG(*).

- The argument of an aggregate function is a set of values derived from an expression. The expression may include columns but cannot include another aggregate function. The scope of the set is a group or an intermediate result table as explained in Chapter 4, "Queries".
- If a GROUP BY clause is specified in a query and the intermediate result of the FROM, WHERE, GROUP BY, and HAVING clauses is the empty set, then the aggregate functions are not applied, the result of the query is the empty set.
- If a GROUP BY clause is not specified in a query and the intermediate result of the FROM, WHERE, and HAVING clauses is the empty set, then the aggregate functions are applied to the empty set. For example, the result of the following SELECT statement is applied to the empty set because department D01 has no employees:

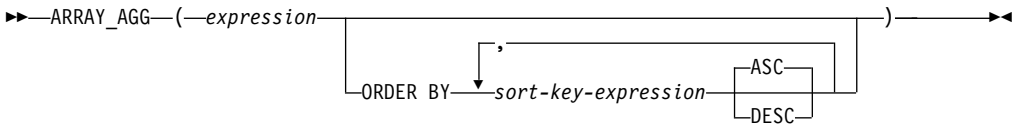
```
SELECT COUNT(DISTINCT JOB)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D01'
```

- The keyword DISTINCT is not considered an argument of the function, but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, redundant duplicate values are eliminated. If ALL is implicitly or explicitly specified, redundant duplicate values are not eliminated.

  When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).
- An aggregate function can be used in a WHERE clause only if that clause is part of a subquery of a HAVING clause and the column name specified in the expression is a correlated reference to a group. If the expression includes more than one column name, each column name must be a correlated reference to the same group.

ARRAY_AGG

The ARRAY_AGG function aggregates a set of elements into an array.



expression
An expression that returns a value that is any data type that can be specified for a CREATE TYPE (Array) statement.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is not specified, or if the ORDER BY clause cannot differentiate the order of the sort key value, the rows in the same grouping set are arbitrarily ordered.

sort-key-expression
Specifies a sort key value that is either a column name or an expression. The data type of the column or expression must not be a DATALINK or XML value.

The ordering of the aggregated elements is based on the values of the sort keys.

The sum of the length attributes of the *sort-key-expressions* must not exceed 3.5 gigabytes.

If a collating sequence other than *HEX is in effect when the statement that contains the ARRAY_AGG function is executed and the *sort-key-expressions* are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values. The weighted values are derived by applying the collating sequence to the *sort-key-expressions*.

The ARRAY_AGG function can only be specified within an SQL procedure in the following specific contexts:

- The select-list of a SELECT INTO statement
- The select-list of a scalar subquery on the right side of a SET statement

The SELECT that uses ARRAY_AGG cannot contain the DISTINCT clause.

Examples

Assume an array type and a table are created as follows:

```
CREATE TYPE PHONELIST AS DECIMAL(10,0) ARRAY[10]

CREATE TABLE EMPLOYEE (
    ID          INTEGER NOT NULL,
    PRIORITY    INTEGER NOT NULL,
    PHONENUMBER DECIMAL(10,0),
    PRIMARY KEY (ID, PRIORITY) )
```

Create a procedure that uses a SELECT INTO statement to return the prioritized list of contact numbers under which an employee can be reached.

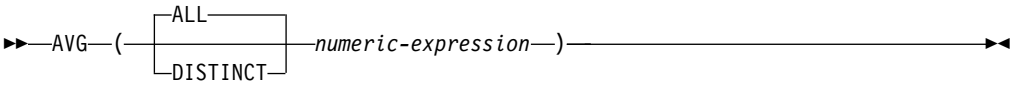
```
CREATE PROCEDURE GETPHONENUMBERS
    (IN EMPID INTEGER,
     OUT NUMBERS PHONELIST)
  BEGIN
    SELECT ARRAY_AGG(PHONENUMBER ORDER BY PRIORITY) INTO NUMBERS
        FROM EMPLOYEE
        WHERE ID = EMPID;
  END
```

Create a procedure that uses a SET statement to return the list of an employee's contact numbers in an arbitrary order.

```
CREATE PROCEDURE GETPHONENUMBERS
    (IN EMPID INTEGER,
     OUT NUMBERS PHONELIST)
  BEGIN
    SET NUMBERS =
      (SELECT ARRAY_AGG(PHONENUMBER)
        FROM EMPLOYEE
        WHERE ID = EMPID);
  END
```

AVG

The AVG function returns the average of a set of numbers.



numeric-expression
The argument values must be any built-in numeric data type and their sum must be within the range of the data type of the result.

The data type of the result is the same as the data type of the argument values, except that:

- The result is DECFLOAT(34) if the argument values are DECFLOAT(16).
- The result is double-precision floating point if the argument values are single-precision floating point.
- The result is large integer if the argument values are small integers.
- The result is decimal if the argument values are decimal or nonzero scale binary with precision *p* and scale *s*. The precision of the result is *p*-*s*+ min(*ms*, *mp*-*p*+*s*). The scale of the result is min(*ms*, *mp*-*p*+*s*).

For information about the values of *p*, *s*, *ms*, and *mp*, see "Decimal arithmetic in SQL" on page 167.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is used, duplicate values are eliminated.

The result can be null. If set of values is empty, the result is the null value. Otherwise, the result is the average value of the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

If the type of the result is integer, the fractional part of the average is lost.

Notes

Results involving DECFLOAT special values: If the data type of the argument is decimal floating-point and a special value of sNaN or -sNaN, or both +Infinity and -Infinity are included in the aggregation, an error or warning is returned. Otherwise, if +NaN or -NaN is found, the result is +NaN or -NaN. If +Infinity or -Infinity is found, the result is +Infinity or -Infinity.

Examples

- Using the PROJECT table, set the host variable AVERAGE (DECIMAL(5,2)) to the average staffing level (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(PRSTAFF)
  INTO :AVERAGE
  FROM PROJECT
  WHERE DEPTNO = 'D11'
```

  Results in AVERAGE being set to 4.25 (that is, 17/4).

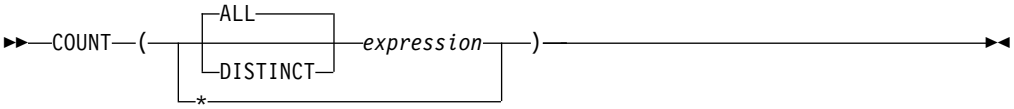
- Using the PROJECT table, set the host variable ANY_CALC to the average of each unique staffing value (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(DISTINCT PRSTAFF)
  INTO :ANY_CALC
  FROM PROJECT
  WHERE DEPTNO = 'D11'
```

  Results in ANY_CALC being set to 4.66 (that is, 14/3).

COUNT

The COUNT function returns the number of rows or values in a set of rows or values.



expression
The argument values can be of any built-in data type other than a DataLink. XML is not allowed for COUNT (DISTINCT *expression*).

The result of the function is a large integer and it must be within the range of large integers. The result cannot be null. If the table is a distributed table, then the result is DECIMAL(15,0). For more information about distributed tables, see the DB2 Multisystem topic collection.

The argument of COUNT(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT(*expression*) or COUNT(ALL *expression*) is a set of values. The function is applied to the set derived from the argument values by the elimination of null values. The result is the number of non-null values in the set including duplicates.

The argument of COUNT(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values and duplicate values. The result is the number of values in the set.

If a collating sequence other than *HEX is in effect when the statement that contains the COUNT(DISTINCT *expression*) is executed and the arguments are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the collating sequence.

Examples

- Using the EMPLOYEE table, set the host variable FEMALE (INTEGER) to the number of rows where the value of the SEX column is 'F'.

```
SELECT COUNT(*)
  INTO :FEMALE
  FROM EMPLOYEE
  WHERE SEX = 'F'
```

  Results in FEMALE being set to 19.

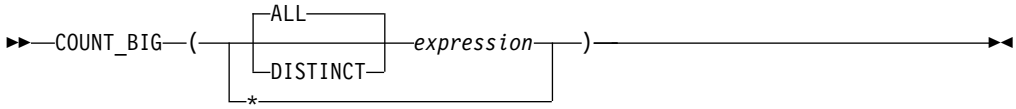
- Using the EMPLOYEE table, set the host variable FEMALE_IN_DEPT (INTEGER) to the number of departments (WORKDEPT) that have at least one female as a member.

```
SELECT COUNT(DISTINCT WORKDEPT)
  INTO :FEMALE_IN_DEPT
  FROM EMPLOYEE
  WHERE SEX='F'
```

  Results in FEMALE_IN_DEPT being set to 6. (There is at least one female in departments A00, C01, D11, D21, E11, and E21.)

COUNT_BIG

The COUNT_BIG function returns the number of rows or values in a set of rows or values. It is similar to COUNT except that the result can be greater than the maximum value of integer.



expression
The argument values can be of any built-in data type other than a DataLink. XML is not allowed for COUNT_BIG(DISTINCT *expression*).

The result of the function is a decimal with precision 31 and scale 0. The result cannot be null.

The argument of COUNT_BIG(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT_BIG(*expression*) is a set of values. The function is applied to the set derived from the argument values by the elimination of null values. The result is the number of values in the set.

If a collating sequence other than *HEX is in effect when the statement that contains the COUNT_BIG(DISTINCT *expression*) is executed and the arguments are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the collating sequence.

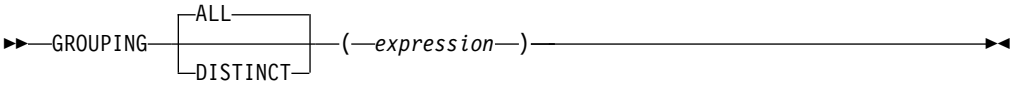
Examples

- Refer to COUNT examples and substitute COUNT_BIG for occurrences of COUNT. The results are the same except for the data type of the result.
- To count on a specific column, a sourced function must specify the type of the column. In this example, the CREATE FUNCTION statement creates a sourced function that takes any column defined as CHAR, uses COUNT_BIG to perform the counting, and returns the result as a double precision floating-point number. The query shown counts the number of unique departments in the sample employee table.

```
CREATE FUNCTION RICK.COUNT(CHAR(19)) RETURNS DOUBLE
  SOURCE QSYS2.COUNT_BIG(CHAR());

SET CURRENT PATH RICK, SYSTEM PATH

SELECT COUNT(DISTINCT WORKDEPT) FROM EMPLOYEE;
```

GROUPING

Used in conjunction with grouping-sets and super-groups, the GROUPING aggregate function returns a value that indicates whether a row returned in a GROUP BY answer set is a row generated by a grouping set that excludes the column represented by *expression*.



expression
The argument values can be any built-in data type, but must be an item of a GROUP BY clause.

The data type of the result is a small integer. It is set to one of the following values:

1 The value of *expression* in the returned row is a null value, and the row was generated by the super-group. This generated row can be used to provide subtotal values for the GROUP BY expression.

0 The value is other than the above.

Example

The following query:

```
SELECT SALES_DATE, SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD,
       GROUPING(SALES_DATE) AS DATE_GROUP,
       GROUPING(SALES_PERSON) AS SALES_GROUP
FROM SALES
GROUP BY CUBE( SALES_DATE, SALES_PERSON)
ORDER BY SALES_DATE, SALES_PERSON
```

Results in:

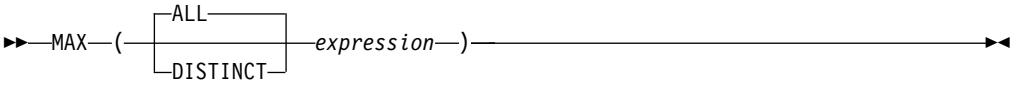
```
SALES_DATE SALES_PERSON UNITS_SOLD DATE_GROUP SALES_GROUP
---------- ------------ ---------- ---------- -----------
12/31/1995 GOUNOT                1          0           0
12/31/1995 LEE                   6          0           0
12/31/1995 LUCCHESSI             1          0           0
12/31/1995 -                     8          0           1
03/29/1996 GOUNOT               11          0           0
03/29/1996 LEE                  12          0           0
03/29/1996 LUCCHESSI             4          0           0
03/29/1996 -                    27          0           1
03/30/1996 GOUNOT               21          0           0
03/30/1996 LEE                  21          0           0
03/30/1996 LUCCHESSI             4          0           0
03/30/1996 -                    46          0           1
03/31/1996 GOUNOT                3          0           0
03/31/1996 LEE                  27          0           0
03/31/1996 LUCCHESSI             1          0           0
03/31/1996 -                    31          0           1
04/01/1996 GOUNOT               14          0           0
04/01/1996 LEE                  25          0           0
04/01/1996 LUCCHESSI             4          0           0
04/01/1996 -                    43          0           1
-          GOUNOT               50          1           0
```

```
-          LEE                91          1          0
-          LUCCHESSI          14          1          0
-          -                 155          1          1
```

An application can recognize a SALES_DATE subtotal row by the fact that the value of DATE_GROUP is 0 and the value of SALES_GROUP is 1. A SALES_PERSON subtotal row can be recognized by the fact that the value of DATE_GROUP is 1 and the value of SALES_GROUP is 0. A grand total row can be recognized by the fact that the value of both DATE_GROUP and SALES_GROUP is 1.

MAX

The MAX aggregate function returns the maximum value in a set of values in a group.



expression
The argument values can be any built-in data type other than a DataLink or XML.

The data type and length attribute of the result are the same as the data type and length attribute of the argument values. When the argument is a string, the result has the same CCSID as the argument.

If a collating sequence other than *HEX is in effect when the statement that contains the MAX function is executed and the arguments are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the collating sequence.

The function is applied to the set of values derived from the argument values by the elimination of null values.

The result can be null. If the function is applied to the empty set, the result is a null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and is not advised.

Notes

Results involving DECFLOAT special values: If the data type of the argument is decimal floating-point and positive or negative Infinity, sNaN, or NaN is found, the maximum value is determined using decimal floating-point ordering rules. See "Numeric comparisons" on page 110. If multiple representations of the same decimal floating-point value are found (for example, 2.00 and 2.0), it is unpredictable which representation will be returned.

Examples

- Using the EMPLOYEE table, set the host variable MAX_SALARY (DECIMAL(7,2)) to the maximum monthly salary (SALARY / 12) value.

```
SELECT MAX(SALARY) /12
  INTO :MAX_SALARY
  FROM EMPLOYEE
```

  Results in MAX_SALARY being set to 4395.83.

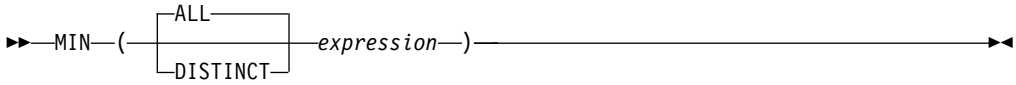
- Using the PROJECT table, set the host variable LAST_PROJ (CHAR(24)) to the project name (PROJNAME) that comes last in the sort sequence.

```
SELECT MAX(PROJNAME)
  INTO :LAST_PROJ
  FROM PROJECT
```

  Results in LAST_PROJ being set to 'WELD LINE PLANNING   '.

MIN

The MIN aggregate function returns the minimum value in a set of values in a group.



expression
The argument values can be any built-in data type other than a DataLink or XML.

The data type and length attribute of the result are the same as the data type and length attribute of the argument values. When the argument is a string, the result has the same CCSID as the argument. The result can be null.

If a collating sequence other than *HEX is in effect when the statement that contains the MIN function is executed and the arguments are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values for each value in the set.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to the empty set, the result is a null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and is not advised.

Notes

Results involving DECFLOAT special values: If the data type of the argument is decimal floating-point and positive or negative Infinity, sNaN, or NaN is found, the minimum value is determined using decimal floating-point ordering rules. See "Numeric comparisons" on page 110. If multiple representations of the same decimal floating-point value are found (for example, 2.00 and 2.0), it is unpredictable which representation will be returned.

Examples

- Using the EMPLOYEE table, set the host variable COMM_SPREAD (DECIMAL(7,2)) to the difference between the maximum and minimum commission (COMM) for the members of department (WORKDEPT) 'D11'.

```
SELECT MAX(COMM) - MIN(COMM)
  INTO :COMM_SPREAD
  FROM EMPLOYEE
  WHERE WORKDEPT  = 'D11'
```

  Results in COMM_SPREAD being set to 1118 (that is, 2580 - 1462).

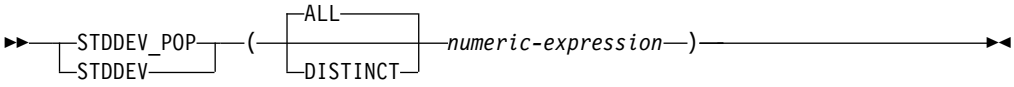
- Using the PROJECT table, set the host variable FIRST_FINISHED (CHAR(10)) to the estimated ending date (PRENDATE) of the first project scheduled to be completed.

```
SELECT MIN(PRENDATE)
  INTO :FIRST_FINISHED
  FROM PROJECT
```

Results in FIRST_FINISHED being set to ‘1982-09-15'.

STDDEV_POP or STDDEV

The STDDEV_POP function returns the biased standard deviation (/n) of a set of numbers.



The formula used to calculate the biased standard deviation is:

```
STDDEV_POP = SQRT(VAR_POP)
```

where SQRT(VAR_POP) is the square root of the variance.

numeric-expression
The argument values must be any built-in numeric data type.

If the argument is DECFLOAT(n), the result of the function is DECFLOAT(34). Otherwise, the data type of the result is double-precision floating point.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, duplicate values are eliminated.

The result can be null. If the function is applied to the empty set, the result is a null value. Otherwise, the result is the standard deviation of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Notes

Results involving DECFLOAT special values: If the data type of the argument is decimal floating-point and a special value of sNaN or -sNaN, or both +Infinity and -Infinity are included in the aggregation, an error or warning is returned. Otherwise, if +NaN or -NaN is found, the result is +NaN or -NaN. If +Infinity or -Infinity is found, the result is +Infinity or -Infinity.

Syntax alternatives: STDEV_POP should be used for conformance to the SQL 2003 standard.

Example

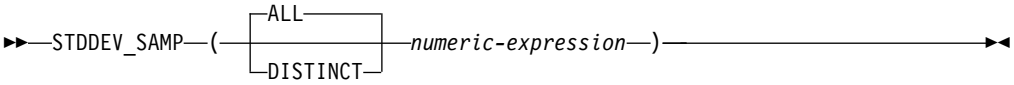
- Using the EMPLOYEE table, set the host variable DEV (double-precision floating point) to the standard deviation of the salaries for those employees in department A00.

```
SELECT STDDEV_POP(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';
```

Results in DEV being set to approximately 9742.43.

STDDEV_SAMP

The STDDEV_SAMP function returns the sample standard deviation (/n-1) of a set of numbers.



The formula used to calculate the sample standard deviation is:

```
STDDEV_SAMP = SQRT(VAR_SAMP)
```

where SQRT(VAR_SAMP) is the square root of the sample variance.

numeric-expression
The argument values must be any built-in numeric data type.

If the argument is DECFLOAT(n), the result of the function is DECFLOAT(34). Otherwise, the data type of the result is double-precision floating point.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, duplicate values are eliminated.

The result can be null. If the function is applied to the empty set or a set with only one row, the result is a null value. Otherwise, the result is the standard deviation of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Notes

Results involving DECFLOAT special values: If the data type of the argument is decimal floating-point and a special value of sNaN or -sNaN, or both +Infinity and -Infinity are included in the aggregation, an error or warning is returned. Otherwise, if +NaN or -NaN is found, the result is +NaN or -NaN. If +Infinity or -Infinity is found, the result is +Infinity or -Infinity.

Example

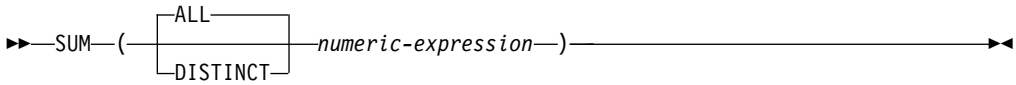
- Using the EMPLOYEE table, set the host variable DEV (double-precision floating point) to the sample standard deviation of the salaries for those employees in department A00.

```
SELECT STDDEV_SAMP(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';
```

Results in DEV being set to approximately 10892.37.

SUM

The SUM function returns the sum of a set of numbers.



numeric-expression
The argument values must be any built-in numeric data type.

The data type of the result is the same as the data type of the argument values except that the result is:

- DECFLOAT(34) if the argument values are DECFLOAT(16).
- A double-precision floating point if the argument values are single-precision floating point
- A large integer if the argument values are small integers
- A decimal with precision *mp* and scale *s* if the argument values are decimal or nonzero scale binary numbers with precision *p* and scale *s*.

For information about the values of *p*, *s*, and *mp*, see "Decimal arithmetic in SQL" on page 167.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, duplicate values are eliminated.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Notes

Results involving DECFLOAT special values: If the data type of the argument is decimal floating-point and a special value of sNaN or -sNaN, or both +Infinity and -Infinity are included in the aggregation, an error is signaled. Otherwise, if +NaN or -NaN is found, the result is +NaN or -NaN. If +Infinity or -Infinity is found, the result is +Infinity or -Infinity.

Example

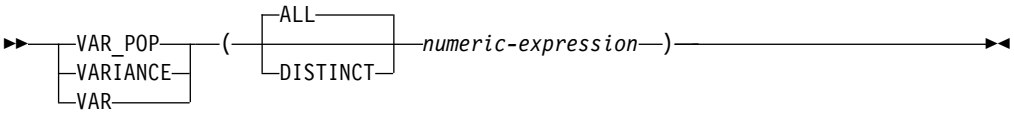
- Using the EMPLOYEE table, set the host variable JOB_BONUS (DECIMAL(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').

```
SELECT SUM(BONUS)
  INTO :JOB_BONUS
  FROM EMPLOYEE
  WHERE JOB = 'CLERK'
```

  Results in JOB_BONUS being set to 4000.

VAR_POP or VARIANCE or VAR

The VAR_POP function returns the biased variance (/n) of a set of numbers.



The formula used to calculate the biased variance is:

```
VAR_POP = SUM(X**2)/COUNT(X) - (SUM(X)/COUNT(X))**2
```

numeric-expression
The argument values must be any built-in numeric data type.

If the argument is DECFLOAT(n), the result of the function is DECFLOAT(34). Otherwise, the data type of the result is double-precision floating point.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, duplicate values are eliminated.

The result can be null. If the function is applied to the empty set, the result is a null value. Otherwise, the result is the variance of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Notes

Results involving DECFLOAT special values: If the data type of the argument is decimal floating-point and a special value of sNaN or -sNaN, or both +Infinity and -Infinity are included in the aggregation, an error or warning is returned. Otherwise, if +NaN or -NaN is found, the result is +NaN or -NaN. If +Infinity or -Infinity is found, the result is +Infinity or -Infinity.

Syntax alternatives: VAR_POP should be used for conformance to the SQL 2003 standard.

Example

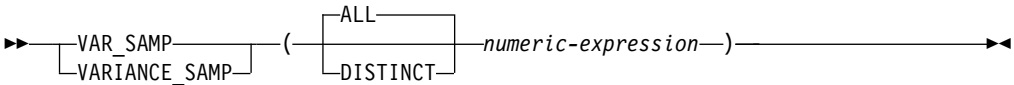
- Using the EMPLOYEE table, set the host variable VARNCE (double-precision floating point) to the variance of the salaries for those employees in department A00.

```
SELECT VAR_POP(SALARY)
  INTO :VARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';
```

  Results in VARNCE being set to approximately 94 915 000.

VARIANCE_SAMP or VAR_SAMP

The VAR_SAMP function returns the sample variance (/n-1) of a set of numbers.



The formula used to calculate the sample variance is:

```
VAR_SAMP = (SUM(X**2) - ((SUM(X)**2) / (COUNT(X)))) / (COUNT(X) - 1)
```

numeric-expression
The argument values must be any built-in numeric data type.

If the argument is DECFLOAT(n), the result of the function is DECFLOAT(34). Otherwise, the data type of the result is double-precision floating point.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, duplicate values are eliminated.

The result can be null. If the function is applied to the empty set or a set with only one row, the result is a null value. Otherwise, the result is the variance of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Notes

Results involving DECFLOAT special values: If the data type of the argument is decimal floating-point and a special value of sNaN or -sNaN, or both +Infinity and -Infinity are included in the aggregation, an error or warning is returned. Otherwise, if +NaN or -NaN is found, the result is +NaN or -NaN. If +Infinity or -Infinity is found, the result is +Infinity or -Infinity.

Syntax alternatives: VAR_SAMP should be used for conformance to the SQL 2003 standard.

Example

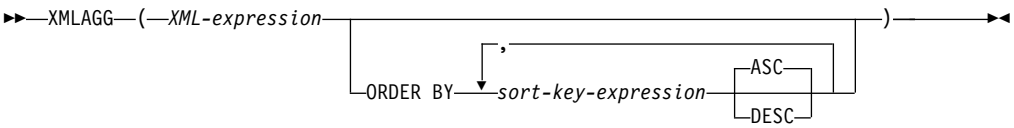
- Using the EMPLOYEE table, set the host variable VARNCE (double-precision floating point) to the sample variance of the salaries for those employees in department A00.

```
SELECT VAR_SAMP(SALARY)
  INTO :VARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';
```

Results in VARNCE being set to approximately 1 186 437 500.

XMLAGG

The XMLAGG function returns an XML sequence containing an item for each non-null value in a set of XML values.



XML-expression
An expression that returns an XML value.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is not specified, or if the ORDER BY clause cannot differentiate the order of the sort key value, the rows in the same grouping set are arbitrarily ordered.

sort-key-expression
Specifies a sort key value that is either a column name or an expression. The data type of the column or expression must not be a DATALINK or XML value.

The ordering is based on the values of the sort keys, which might or might not be used in *XML-expression*.

The sum of the length attributes of the *sort-key-expressions* must not exceed 3.5 gigabytes.

If a collating sequence other than *HEX is in effect when the statement that contains the XMLAGG function is executed and the *sort-key-expressions* are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values. The weighted values are derived by applying the collating sequence to the *sort-key-expressions*.

The function is applied to the set of values derived from the argument values by elimination of null values.

The data type of the result is XML. The result can be null. If the function is applied to an empty set, the result is the null value. Otherwise, the result is an XML sequence containing an item for each value in the set.

Example

Note: XMLAGG does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Group employees by their department, generate a "Department" element for each department with its name as the attribute, nest all the "emp" elements for employees in each department, and order the "emp" elements by LASTNAME.

```
SELECT XMLSERIALIZE(XMLDOCUMENT (
            XMLELEMENT(NAME "Department",
                       XMLATTRIBUTES(E.WORKDEPT AS "name"),
                       XMLAGG(XMLELEMENT ( NAME "emp", E.LASTNAME)
                              ORDER BY E.LASTNAME)
                       ))
```

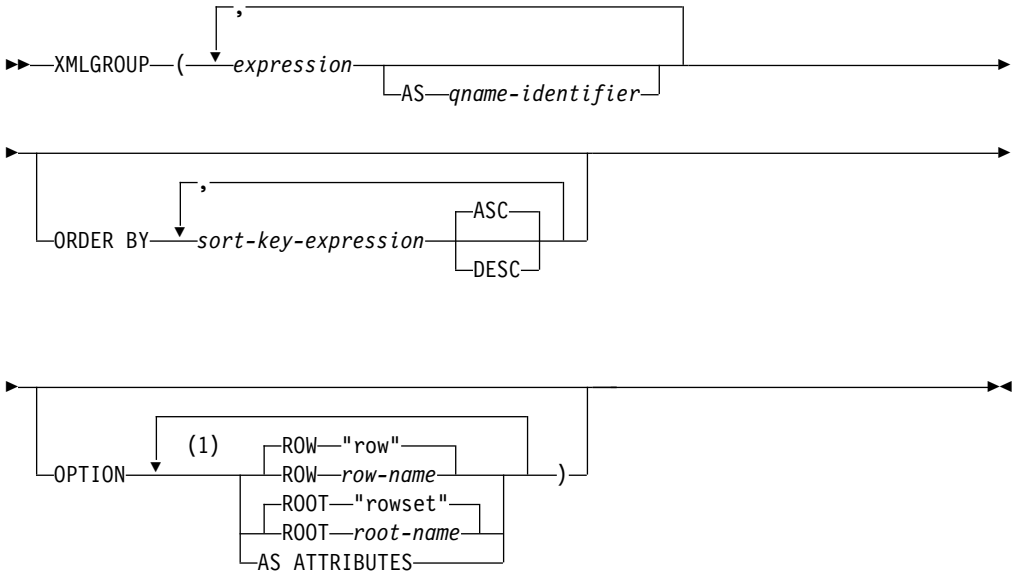
```
                          AS CLOB(200)) AS "dept_list"
  FROM EMPLOYEE E
  WHERE E.WORKDEPT IN ('C01', 'E21')
  GROUP BY WORKDEPT
```

The result of the query would look similar to the following result:

```
dept_list
---------------------------------
<Department name="C01">
  <emp>KWAN</emp>
  <emp>NICHOLLS</emp>
  <emp>QUINTANA</emp>
</Department>
<Department name="E21">
  <emp>GOUNOT</emp>
  <emp>LEE</emp>
  <emp>MEHTA</emp>
  <emp>SPENSER</emp>
</Department>
```

XMLGROUP

The XMLGROUP function returns an XML value that is a well-formed XML document.



Notes:

1 The same clause must not be specified more than once.

expression
The content of each XML element is specified by an expression. The data type of *expression* must not be ROWID or DATALINK or a distinct type that is based on ROWID or DATALINK. The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified. When AS ATTRIBUTES is specified, the data type of *expression* must not be XML or a distinct type that is based on XML.

AS *qname-identifier*
Specifies the XML element name or attribute name as an SQL identifier. The *qname-identifier* must be of the form of an XML qualified name, or QName. See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope. If *qname-identifier* is not specified, *expression* must be a column name. The element name or attribute name is created from the column name using the fully escaped mapping from a column name to a QName.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is not specified, or if the ORDER BY clause cannot differentiate the order of the sort key value, the rows in the same grouping set are arbitrarily ordered.

sort-key-expression
Specifies a sort key value that is either a column name or an expression. The data type of the column or expression must not be a DATALINK or XML value.

The ordering is based on the values of the sort keys, which might or might not be used in *XML-expression*.

OPTION
Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

ROW *row-name*
Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

ROOT *root-name*
Specifies the name of the root element. If this option is not specified, the default root element name is "rowset".

AS ATTRIBUTES
Specifies that each expression is mapped to an attribute value with column name or *qname-identifier* serving as the attribute name.

If a collating sequence other than *HEX is in effect when the statement that contains the XMLGROUP function is executed and the *sort-key-expressions* are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values. The weighted values are derived by applying the collating sequence to the *sort-key-expressions*.

The result of the function is XML. The result can be null. If the set of values is empty, the result is the null value. Otherwise, the result is an XML sequence containing an item for each value in the set.

Notes

The default behavior defines a simple mapping between a result set and an XML value Some additional notes about function behavior apply:

- By default, each row is transformed into an XML element named "row" and each *expression* is transformed into a nested element with the column name or *qname-identifier* as the element name.
- The null handling behavior is NULL ON NULL. A null value for an expression maps to the absence of the subelement. If all expression values are null, no row element is generated. If no row elements are generated, the function returns the null value.
- The binary encoding scheme for binary and FOR BIT DATA data types is base64Binary encoding.
- The order of the row subelements in the root element will be the same as the order in which the rows are returned in the query result set.

Examples

Note: XMLGROUP does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

Assume the following table T1 with columns C1 and C2:

```
 C1    C2
----  ----
  1     2
  -     2
  1     -
  -     -
```

- The following example shows an XMLGROUP query and output fragment with default behavior, using a single top-level element to represent the table:

```
SELECT XMLGROUP(C1, C2) FROM T1
```

  Returns the following value for the single result row:

```
<rowset>
<row><C1>1</C1><C2>2</C2></row>
<row><C2>2</C2></row>
<row><C1>1</C1></row>
</rowset>
```

- The following example shows an XMLGROUP query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, the data is mapped to element attributes:

```
SELECT XMLGROUP(C1, C2 OPTION AS ATTRIBUTES) FROM T1
```

  Returns the following value for the single result row:

```
<rowset>
<row C1="1" C2="2"/>
<row C2="2"/>
<row C1="1"/>
</rowset>
```

- The following example shows an XMLGROUP query and output fragment with the default <rowset> root element replaced by <document> and the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the return set is ordered by column C1:

```
SELECT XMLGROUP(C1 AS "column1", C2 AS "column2"
                ORDER BY C1 OPTION ROW "entry" ROOT "document")
       FROM T1
```

  Returns the following value for the single result row:

```
<document>
<entry> <column1>1</column1><column2>2</column2></entry>
<entry> <column1>1</column1></entry>
<entry> <column2>2</column2></entry>
</document>
```

Scalar functions

A *scalar function* takes input argument(s) and returns a single value result. A scalar function can be used wherever an expression can be used.

The restrictions on the use of aggregate functions do not apply to scalar functions, because a scalar function is applied to single parameter values rather than to sets of values. The argument of a scalar function can be a function. However, the restrictions that apply to the use of expressions and aggregate functions also apply when an expression or aggregate function is used within a scalar function. For example, the argument of a scalar function can be an aggregate function only if an aggregate function is allowed in the context in which the scalar function is used.

Example

The result of the following SELECT statement has as many rows as there are employees in department D01:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BIRTHDATE)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D01'
```

ABS

The ABS function returns the absolute value of a number.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: For decimal floating-point values, the special values are treated as follows:

- ABS(NaN) and ABS(-NaN) return NaN.
- ABS(Infinity) and ABS(-Infinity) return Infinity.
- ABS(sNaN) and ABS(-sNaN) return sNaN.

Syntax alternatives: ABSVAL is a synonym for ABS. It is supported only for compatibility with previous DB2 releases.

Example

- Assume the host variable PROFIT is a large integer with a value of -50000.

```
SELECT ABS(:PROFIT)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 50000.

ACOS

The ACOS function returns the arc cosine of the argument as an angle expressed in radians. The ACOS and COS functions are inverse operations.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. The value must be greater than or equal to -1 and less than or equal to 1.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is greater than or equal to 0 and less than or equal to π.

Example

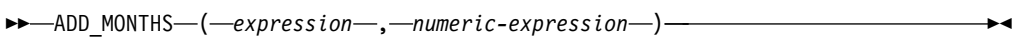
- Assume the host variable ACOSINE is a DECIMAL(10,9) host variable with a value of 0.070737202.

```
SELECT ACOS(:ACOSINE)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 1.49.

ADD_MONTHS

The ADD_MONTHS function returns a date or timestamp that represents *expression* plus *numeric-expression* months.



expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

numeric-expression
An expression that returns a value of a built-in numeric data type with zero scale. A negative numeric value is allowed.

The result of the function is a timestamp if *expression* is a timestamp. Otherwise, the result of the function is a date. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

If *expression* is the last day of the month or if the resulting month has fewer days than the day component of *expression*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *expression*.

Example

- Assume today is January 31, 2000. Set the host variable ADD_MONTH with the last day of January plus 1 month.

  ```
  SET :ADD_MONTH = ADD_MONTHS(LAST_DAY(CURRENT_DATE), 1 )
  ```

  The host variable ADD_MONTH is set with the value representing the end of February, 2000-02-29.
- Assume DATE is a host variable with the value July 27, 1965. Set the host variable ADD_MONTH with the value of that day plus 3 months.

  ```
  SET :ADD_MONTH = ADD_MONTHS(:DATE, 3)
  ```

  The host variable ADD_MONTH is set with the value representing the day plus 3 months, 1965-10-27.
- It is possible to achieve similar results with the ADD_MONTHS function and date arithmetic. The following examples demonstrate the similarities and contrasts.

  ```
  SET :DATEHV = DATE('2000-2-28') + 4 MONTHS
  SET :DATEHV ADD_MONTHS('2000-2-28', 4)
  ```

  In both cases, the host variable DATEHV is set with the value '2000–06–28'.

  Now consider the same examples but with the date '2000–2–29' as the argument.

  ```
  SET :DATEHV = DATE('2000-2-29') + 4 MONTHS
  ```

  The host variable DATEHV is set with the value '2000–06–29'.

  ```
  SET :DATEHV ADD_MONTHS('2000-2-29', 4)
  ```

  The host variable DATEHV is set with the value '2000–06–30'.

  In this case, the ADD_MONTHS function returns the last day of the month, which is June 30, 2000, instead of June 29, 2000. The reason is that February 29 is the last day of the month. So, the ADD_MONTHS function returns the last day of June.

ANTILOG

The ANTILOG function returns the anti-logarithm (base 10) of a number. The ANTILOG and LOG functions are inverse operations.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

If the data type of the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: For decimal floating-point values the special values are treated as follows:

- ANTILOG(NaN) returns NaN.
- ANTILOG(-NaN) returns -NaN.
- ANTILOG(Infinity) returns Infinity.
- ANTILOG(-Infinity) returns 0.
- ANTILOG(sNaN) and ANTILOG(-sNaN) return a warning or error. [49]

Example

- Assume the host variable ALOG is a DECIMAL(10,9) host variable with a value of 1.499961866.

```
SELECT ANTILOG(:ALOG)
  FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 31.62.

49. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN and -NaN are returned respectively with a warning.

ASCII

The ASCII function returns the ASCII code value of the leftmost character of the argument as an integer.

expression
An expression that specifies the string containing the character to evaluate. *expression* must be any built-in numeric or string data type. The first character of the string will be converted to ASCII CCSID 367 for processing by the function.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Return the integer value for the ASCII representation of 'A'.

```
SELECT ASCII('A')
    FROM SYSIBM.SYSDUMMY1
```

  Returns the value 65.

ASIN

The ASIN function returns the arc sine of the argument as an angle expressed in radians. The ASIN and SIN functions are inverse operations.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. The value must be greater than or equal to -1 and less than or equal to 1.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is greater than or equal to $-\pi/2$ and less than or equal to $\pi/2$.

Example

- Assume the host variable ASINE is a DECIMAL(10,9) host variable with a value of 0.997494987.

```
SELECT ASIN(:ASINE)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 1.50.

ATAN

The ATAN function returns the arc tangent of the argument as an angle expressed in radians. The ATAN and TAN functions are inverse operations.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is greater than or equal to $-\pi/2$ and less than or equal to $\pi/2$.

Example

- Assume the host variable ATANGENT is a DECIMAL(10,8) host variable with a value of 14.10141995.

```
SELECT ATAN(:ATANGENT)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 1.50.

ATANH

The ATANH function returns the hyperbolic arc tangent of a number, in radians. The ATANH and TANH functions are inverse operations.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. The value must be greater than -1 and less than 1.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

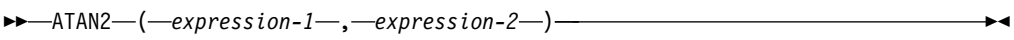
- Assume the host variable HATAN is a DECIMAL(10,9) host variable with a value of 0.905148254.

```
SELECT ATANH(:HATAN)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 1.50.

ATAN2

The ATAN2 function returns the arc tangent of x and y coordinates as an angle expressed in radians. The first and second arguments specify the x and y coordinates, respectively.



expression-1
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information on converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. If one argument is 0, the other argument must not be 0.

expression-2
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information on converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. If one argument is 0, the other argument must not be 0.

The data type of the result is double-precision floating point. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Example

- Assume that host variables HATAN2A and HATAN2B are DOUBLE host variables with values of 1 and 2, respectively.

```
SELECT ATAN2(:HATAN2A,:HATAN2B)
  FROM SYSIBM.SYSDUMMY1
```

  Returns a double precision floating-point number with an approximate value of 1.1071487.

BIGINT

The BIGINT function returns a big integer representation.

Numeric to Big Integer

String to Big Integer

The BIGINT function returns a big integer representation of:

- A number
- A character or graphic string representation of a decimal number
- A character or graphic string representation of an integer
- A character or graphic string representation of a floating-point number
- A character or graphic string representation of a decimal floating-point number

Numeric to Big Integer

numeric-expression
An expression that returns a numeric value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of big integers, an error is returned. The fractional part of the argument is truncated.

String to Big Integer

string-expression
An expression that returns a value that is a character-string or graphic-string representation of a number.

The result is the same number that would result from CAST(*string-expression* AS BIGINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant. If the whole part of the argument is not within the range of big integers, an error is returned. Any fractional part of the argument is truncated.

The result of the function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

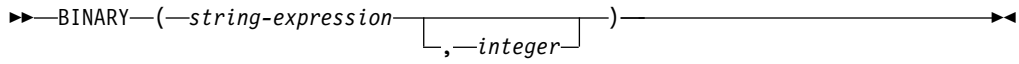
Example

- Using the EMPLOYEE table, select the SALARY column in big integer form for further processing in the application.

```
SELECT BIGINT(SALARY)
  FROM EMPLOYEE
```

BINARY

The BINARY function returns a BINARY representation of a string of any type.



The result of the function is a fixed-length binary string. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

string-expression
A *string-expression* whose value must be a built-in character string, graphic string, binary string, or row ID data type.

integer
An integer constant that specifies the length attribute for the resulting binary string. The value must be between 1 and 32766.

If *integer* is not specified:

- If the *string-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument, unless the argument is a graphic string. In this case, the length attribute of the result is twice the length attribute of the argument.

The actual length is the same as the length attribute of the result. If the length of the *string-expression* is less than the length of the result, the result is padded with hexadecimal zeroes up to the length of the result. If the length of the *string-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the first input argument is a character string and all the truncated characters are blanks, or the first input argument is a graphic string and all the truncated characters are double-byte blanks, or the first input argument is a binary string and all the truncated bytes are hexadecimal zeroes.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the length is specified. For more information, see "CAST specification" on page 185.

Example

- The following function returns a BINARY for the string 'This is a BINARY'.

```
SELECT BINARY('This is a BINARY')
  FROM SYSIBM.SYSDUMMY1
```

BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT

These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.

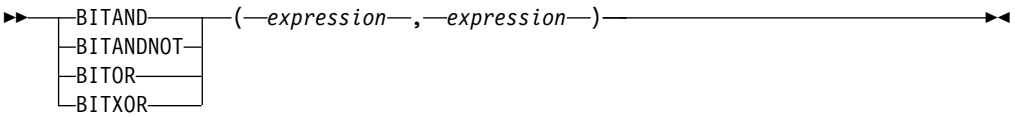


Table 46. Bit manipulation functions

Function	Description	A bit in the two's complement representation of the result is:
BITAND	Performs a bitwise AND operation.	1 only if the corresponding bits in both arguments are 1.
BITANDNOT	Clears any bit in the first argument that is in the second argument.	Zero if the corresponding bit in the second argument is 1; otherwise, the result is copied from the corresponding bit in the first argument.
BITOR	Performs a bitwise OR operation.	1 unless the corresponding bits in both arguments are zero.
BITXOR	Performs a bitwise exclusive OR operation.	1 unless the corresponding bits in both arguments are the same.
BITNOT	Performs a bitwise NOT operation.	Opposite of the corresponding bit in the argument.

expression
An expression that returns a value of any built-in numeric data type. Arguments of type DECIMAL, REAL, or DOUBLE are cast to DECFLOAT. The value is truncated to a whole number.

The bit manipulation functions can operate on up to 16 bits for SMALLINT, 32 bits for INTEGER, 64 bits for BIGINT, and 113 bits for DECFLOAT. The range of supported DECFLOAT values includes integers from -2^{112} to 2^{112} -1, and special values such as NaN or INFINITY are not supported. If the two arguments have different data types, the argument supporting fewer bits is cast to a value with the data type of the argument supporting more bits. This cast impacts the bits that are set for negative values. For example, -1 as a SMALLINT value has 16 bits set to 1, which when cast to an INTEGER value has 32 bits set to 1.

The result of the functions with two arguments has the data type of the argument that is highest in the data type precedence list for promotion. If either argument is DECFLOAT, the data type of the result is DECFLOAT(34). If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The result of the BITNOT function has the same data type as the input argument, except that DECIMAL, REAL, DOUBLE, or DECFLOAT(16) returns DECFLOAT(34). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Use of the BITXOR function is recommended to toggle bits in a value. Use the BITANDNOT function to clear bits. BITANDNOT(val, pattern) operates more efficiently than BITAND(val, BITNOT(pattern)).

Example

The following examples are based on an ITEM table with a PROPERTIES column of type INTEGER.

- Return all items for which the third property bit is set.

```
SELECT ITEMID FROM ITEM
 WHERE BITAND(PROPERTIES, 4) = 4
```

- Return all items for which the fourth or the sixth property bit is set.

```
SELECT ITEMID FROM ITEM
 WHERE BITAND(PROPERTIES, 40) <> 0
```

- Clear the twelfth property of the item whose ID is 3412.

```
UPDATE ITEM
 SET PROPERTIES = BITANDNOT(PROPERTIES, 2048)
 WHERE ITEMID = 3412
```

- Set the fifth property of the item whose ID is 3412.

```
UPDATE ITEM
 SET PROPERTIES = BITOR(PROPERTIES, 16)
 WHERE ITEMID = 3412
```

- Toggle the eleventh property of the item whose ID is 3412.

```
UPDATE ITEM
 SET PROPERTIES = BITXOR(PROPERTIES, 1024)
 WHERE ITEMID = 3412
```

- Switch all the bits in a 16-bit value that has only the second bit on.

```
VALUES BITNOT(CAST(2 AS SMALLINT))
```

  returns -3 (with a data type of SMALLINT).

BIT_LENGTH

The BIT_LENGTH function returns the length of a string expression in bits.

See "LENGTH" on page 392, "CHARACTER_LENGTH" on page 285, and "OCTET_LENGTH" on page 437 for similar functions.

expression
An expression that returns a value of any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The result of the function is DECIMAL(31). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is the number of bits (bytes * 8) in the argument. The length of a string includes trailing blanks. The length of a varying-length string is the actual length in bits (bytes * 8), not the maximum length.

Example

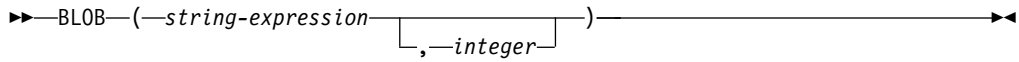
- Assume table T1 has a GRAPHIC(10) column called C1.

```
SELECT BIT_LENGTH( C1 )
  FROM T1
```

  Returns the value 160.

BLOB

The BLOB function returns a BLOB representation of a string of any type.



string-expression
A *string-expression* whose value can be a character string, graphic string, binary string, or row ID.

integer
An integer constant that specifies the length attribute for the resulting binary string. The value must be between 1 and 2 147 483 647.

If *integer* is not specified:

- If the *string-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument, unless the argument is a graphic string. In this case, the length attribute of the result is twice the length attribute of the argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of the expression (or twice the length of the expression when the input is graphic data). If the length of the *string-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the first input argument is a character string and all the truncated characters are blanks, or the first input argument is a graphic string and all the truncated characters are double-byte blanks, or the first input argument is a binary string and all the truncated bytes are hexadecimal zeroes.

The result of the function is a BLOB. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the length is specified. For more information, see "CAST specification" on page 185.

Example

- The following function returns a BLOB for the string 'This is a BLOB'.

```
SELECT BLOB('This is a BLOB')
  FROM SYSIBM.SYSDUMMY1
```

- The following function returns a BLOB for the large object that is identified by locator myclob_locator.

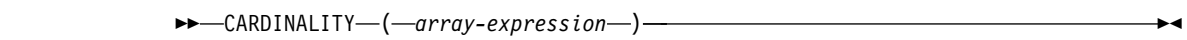
```
SELECT BLOB(:myclob_locator)
  FROM SYSIBM.SYSDUMMY1
```

- Assume that a table has a BLOB column named TOPOGRAPHIC_MAP and a VARCHAR column named MAP_NAME. Locate any maps that contain the string 'Pellow Island' and return a single binary string with the map name concatenated in front of the actual map. The following function returns a BLOB for the large object that is identified by locator myclob_locator.

```
SELECT BLOB( MAP_NAME CONCAT  ':  ' CONCAT  TOPOGRAPHIC_MAP )
  FROM ONTARIO_SERIES_4
  WHERE TOPOGRAPHIC_MAP LIKE '%Pellow Island%'
```

CARDINALITY

The CARDINALITY function returns a value representing the number of elements of an array.



array-expression
The expression can be either an SQL procedure variable or parameter of an array data type, or a cast specification of a parameter marker to an array data type.

The value returned by the CARDINALITY function is the highest subindex for which the array has an assigned element. This includes elements that have been assigned the null value.

The result of the function is BIGINT. The function returns 0 if the array is empty. The result can be null; if the argument is null, the result is the null value.

Example

Assume that type INT_ARRAY is defined as:

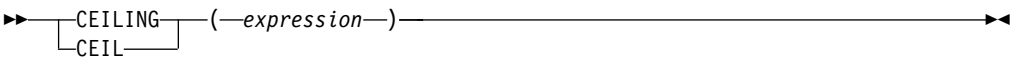
```
CREATE TYPE INT_ARRAY AS INTEGER ARRAY[100]
```

The SET statement in the following code fragment assigns variable LEN the value 4.

```
BEGIN
  DECLARE LEN INTEGER;
  DECLARE MYARRAY INT_ARRAY;

  SET MYARRAY = ARRAY[0,0,1,1];
  SET LEN = CARDINALITY(MYARRAY);
END
```

CEILING

The CEIL or CEILING function returns the smallest integer value that is greater than or equal to *expression*.



expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is a decimal number. For example, an argument with a data type of DECIMAL(5,5) will result in DECIMAL(5,0).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: For decimal floating-point values, the special values are treated as follows:

- CEILING(NaN) returns NaN.
- CEILING(-NaN) returns -NaN.
- CEILING(Infinity) returns Infinity.
- CEILING(-Infinity) returns -Infinity.
- CEILING(sNaN) and CEILING(-sNaN) return a warning or error.[50]

Examples

- Find the highest monthly salary for all the employees. Round the result up to the next integer. The SALARY column has a decimal data type

```
SELECT CEIL(MAX(SALARY)/12)
  FROM EMPLOYEE
```

  This example returns 4396.00 because the highest paid employee is Christine Haas who earns $52750.00 per year. Her average monthly salary before applying the CEIL function is 4395.83.

- Use CEILING on both positive and negative numbers.

```
SELECT CEILING( 3.5),
       CEILING( 3.1),
       CEILING(-3.1),
       CEILING(-3.5)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns:

```
04.   04.  -03.  -03.
```

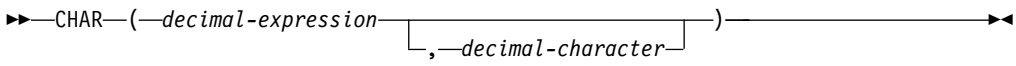
50. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN and -NaN are returned respectively with a warning.

CHAR

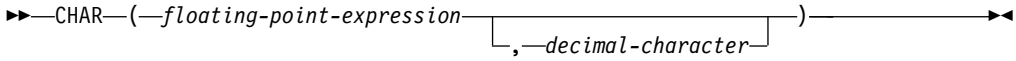
The CHAR function returns a fixed-length character-string representation.

Integer to Character

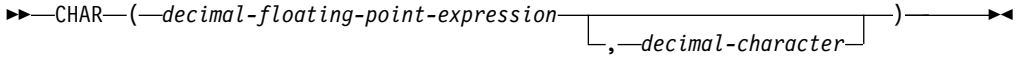
Decimal to Character



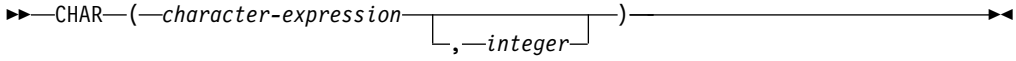
Floating-point to Character



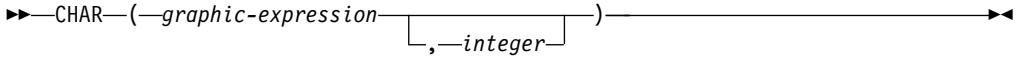
Decimal floating-point to Character



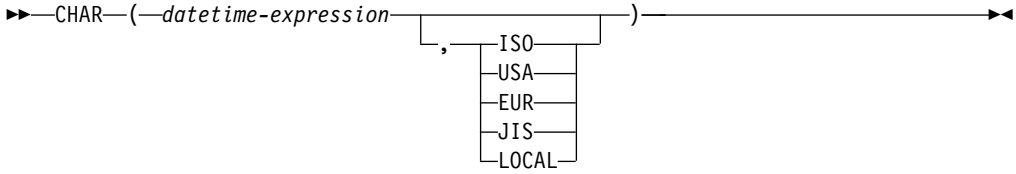
Character to Character



Graphic to Character



Datetime to Character



The CHAR function returns a fixed-length character-string representation of:

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT.
- A decimal number if the first argument is a decimal number.
- A double-precision floating-point number if the first argument is a DOUBLE or REAL.
- A decimal floating-point number if the first argument is a DECFLOAT.

- A character string if the first argument is any type of character string.
- A graphic string if the first argument is any type of graphic string.
- A date value if the first argument is a DATE.
- A time value if the first argument is a TIME.
- A timestamp value if the first argument is a TIMESTAMP.
- A row ID value if the first argument is a ROWID.

The result of the function is a fixed-length character string. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Integer to Character

integer-expression
An expression that returns a value that is a built-in integer data type (either SMALLINT, INTEGER, or BIGINT).

The result is the fixed-length character-string representation of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. The result is left justified.

- If the argument is a small integer:

  The length of the result is 6. If the number of characters in the result is less than 6, then the result is padded on the right with blanks.
- If the argument is a large integer:

  The length of the result is 11. If the number of characters in the result is less than 11, then the result is padded on the right with blanks.
- If the argument is a big integer:

  The length of the result is 20. If the number of characters in the result is less than 20, then the result is padded on the right with blanks.

The CCSID of the string is the default SBCS CCSID at the current server.

Decimal to Character

decimal-expression
An expression that returns a value that is a built-in decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is wanted, the DECIMAL scalar function can be used to make the change.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a fixed-length character string representation of the argument. The result includes a decimal character and up to *p* digits, where *p* is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length of the result is 2+*p* where *p* is the precision of the *decimal-expression*. This means that a positive value will always include one trailing blank.

The CCSID of the string is the default SBCS CCSID at the current server.

Floating-point to Character

floating-point expression
An expression that returns a value that is a built-in floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a fixed-length character-string representation of the argument in the form of a floating-point constant. The length of the result is 24. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0E0. Otherwise, the result includes the smallest number of characters that can be used to represent the value of the argument such that the mantissa consists of a single digit other than zero followed by a *decimal-character* and a sequence of digits.

If the number of characters in the result is less than 24, then the result is padded on the right with blanks.

The CCSID of the string is the default SBCS CCSID at the current server.

Decimal floating-point to Character

decimal-floating-point expression
An expression that returns a value that is a built-in decimal floating-point data type (DECFLOAT).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a fixed-length character-string representation of the argument in the form of a decimal floating-point constant.

The length attribute of the result is 42. The actual length of the result is the smallest number of characters that represents the value of the argument, including the sign, digits, and *decimal-character*. Trailing zeros are significant. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0.

If the DECFLOAT value is Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, a minus sign will be the first character in the string. The DECFLOAT special value sNaN does not result in an exception when converted to a string.

If the number of characters in the result is less than 42, then the result is padded on the right with blanks.

The CCSID of the string is the default SBCS CCSID at the current server.

Character to Character

character-expression
An expression that returns a value that is a built-in character-string data type.[51]

integer
An integer constant that specifies the length attribute for the resulting fixed length character string. The value must be between 1 and 32766 (32765 if nullable). If the first argument is mixed data, the second argument cannot be less than 4.

If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length is the same as the length attribute of the result. If the length of the *character-expression* is less than the length of the result, the result is padded with blanks up to the length of the result. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

The CCSID of the string is the CCSID of the *character-expression*.

Graphic to Character

graphic-expression
An expression that returns a value that is a built-in graphic-string data type.

integer
An integer constant that specifies the length attribute for the resulting fixed length character string. The value must be between 1 and 32766 (32765 if nullable).

If the second argument is not specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 1.
- If the result is SBCS data, the result length is n.
- If the result is mixed data, the result length is $(2.5*(n-1)) + 4$.

The actual length is the same as the length attribute of the result. If the length of the *graphic-expression* is less than the length of the result, the result is padded with blanks up to the length of the result. If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

The CCSID of the string is the default CCSID of the current server.

Datetime to Character

datetime-expression
An expression that is one of the following three built-in data types

date The result is the character-string representation of the date in the format specified by the second argument. If the second argument is not

51. A binary string is also allowed if a CCSID is not specified or if a CCSID of 65535 is explicitly specified.

specified, the format used is the default date format. If the format is ISO, USA, EUR, or JIS, the length of the result is 10. Otherwise the length of the result is the length of the default date format. For more information see "String representations of datetime values" on page 83.

time The result is the character-string representation of the time in the format specified by the second argument. If the second argument is not specified, the format used is the default time format. The length of the result is 8. For more information see "String representations of datetime values" on page 83.

timestamp
The second argument is not applicable and must not be specified.

The result is the character-string representation of the timestamp. The length of the result is 26.

The CCSID of the string is the default SBCS CCSID at the current server.

ISO, EUR, USA, or JIS
Specifies the date or time format of the resulting character string. For more information, see "String representations of datetime values" on page 83.

LOCAL
Specifies that the date or time format of the resulting character string should come from the DATFMT, DATSEP, TIMFMT, and TIMSEP attributes of the job at the current server.

The CCSID of the string is the default SBCS CCSID at the current server.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the first argument is a string and the length argument is specified. For more information, see "CAST specification" on page 185.

Examples

- Assume the column PRSTDATE has an internal value equivalent to 1988-12-25. The date format is *MDY and the date separator is a slash (/).

```
SELECT CHAR(PRSTDATE, USA)
  FROM PROJECT
```

Results in the value '12/25/1988'.

```
SELECT CHAR(PRSTDATE)
  FROM PROJECT
```

Results in the value '12/25/88'.

- Assume the column STARTING has an internal value equivalent to 17.12.30, the host variable HOUR_DUR (DECIMAL(6,0)) is a time duration with a value of 050000 (that is, 5 hours).

```
SELECT CHAR(STARTING, USA)
  FROM CL_SCHED
```

Results in the value '5:12 PM'.

```
SELECT CHAR(STARTING + :HOUR_DUR, JIS)
  FROM CL_SCHED
```

Results in the value '10:12:00'.

- Assume the column RECEIVED (timestamp) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns.

```
SELECT CHAR(RECEIVED)
  FROM IN_TRAY
```

  Results in the value ‘1988-12-25-17.12.30.000000'.
- Use the CHAR function to make the type fixed-length character and reduce the length of the displayed results to 10 characters for the LASTNAME column (defined as VARCHAR(15)) of the EMPLOYEE table.

```
SELECT CHAR(LASTNAME,10)
  FROM EMPLOYEE
```

  For rows having a LASTNAME with a length greater than 10 characters (excluding trailing blanks), a warning (SQLSTATE 01004) that the value is truncated is returned.
- Use the CHAR function to return the values for EDLEVEL (defined as SMALLINT) as a fixed length string.

```
SELECT CHAR(EDLEVEL)
  FROM EMPLOYEE
```

  An EDLEVEL of 18 would be returned as the CHAR(6) value '18 ' (18 followed by 4 blanks).
- Assume that the same SALARY subtracted from 20000.25 is to be returned with a comma as the decimal character.

```
SELECT CHAR(20000.25 - SALARY, ',')
  FROM EMPLOYEE
```

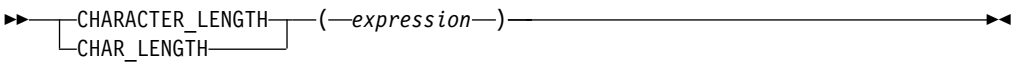
  A SALARY of 21150 returns the value '–1149,75 ' (–1149,75 followed by 3 blanks).
- Assume a host variable, DOUBLE_NUM, has a double precision floating-point data type and a value of -987.654321E-35.

```
SELECT CHAR(:DOUBLE_NUM)
  FROM SYSIBM.SYSDUMMY1
```

  Results in the character value '-9.8765432100000002E-33 '.

CHARACTER_LENGTH

The CHARACTER_LENGTH or CHAR_LENGTH function returns the length of a string expression.



See "LENGTH" on page 392 for a similar function.

expression
An expression that returns a value of any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If *expression* is a character string or graphic string, the result is the number of characters in the argument (not the number of bytes). A single character is either an SBCS, DBCS, or multiple-byte character. If *expression* is a binary string, the result is the number of bytes in the argument. The length of strings includes trailing blanks or hexadecimal zeroes. The length of a varying-length string is the actual length, not the maximum length.

Example

- Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'.

```
SELECT CHARACTER_LENGTH(NAME), LENGTH(NAME)
  FROM T1
  WHERE NAME = 'Jürgen'
```

  Returns the value 6 for CHARACTER_LENGTH and 7 for LENGTH.

CHR

The CHR function returns the character that has the ASCII code value specified by the argument. If expression is 0, the result is the blank character (X'20').

expression
An expression that returns a value of a BIGINT, INTEGER, or SMALLINT data type. The value of the argument should be between 0 and 255; otherwise, the return value is null. The value is interpreted as the code point for a character in ASCII CCSID 367.

The result of the function is CHAR(1). The result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the default SBCS CCSID of the current server.

Examples

- Return the character corresponding to the ASCII CCSID 367 code point 65.

```
SELECT CHR(65)
    FROM SYSIBM.SYSDUMMY1
```

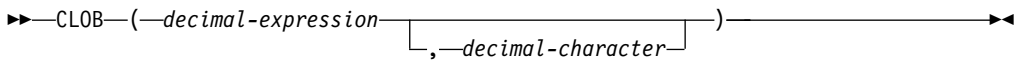
  Returns the value 'A'.

CLOB

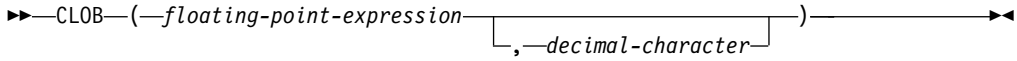
The CLOB function returns a character-string representation.

Integer to CLOB

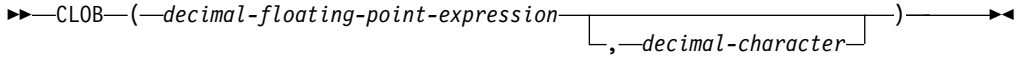
Decimal to CLOB



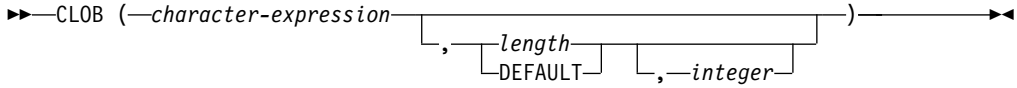
Floating-point to CLOB



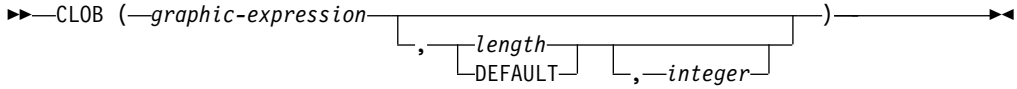
Decimal floating-point to CLOB



Character to CLOB



Graphic to CLOB



The CLOB function returns a character-string representation of:

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number if the first argument is a packed or zoned decimal number
- A double-precision floating-point number if the first argument is a DOUBLE or REAL
- A decimal floating-point number if the first argument is DECFLOAT
- A character string if the first argument is any type of character string
- A graphic string if the first argument is a Unicode graphic string

The result of the function is a CLOB. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Integer to CLOB

integer-expression
An expression that returns a value that is a built-in integer data type (either SMALLINT, INTEGER, or BIGINT).

The result is a varying-length character string of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. The result is left justified.

- If the argument is a small integer, the length attribute of the result is 6.
- If the argument is a large integer, the length attribute of the result is 11.
- If the argument is a big integer, the length attribute of the result is 20.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit or the *decimal-character*.

The CCSID of the result is the default SBCS CCSID at the current server.

Decimal to CLOB

decimal-expression
An expression that returns a value that is a built-in decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is wanted, the DECIMAL scalar function can be used to make the change.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length character string representation of the argument. The result includes a decimal character and up to *p* digits, where *p* is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length attribute of the result is 2+*p* where *p* is the precision of the *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing characters are included. Leading zeros are not included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit or the *decimal-character*.

The CCSID of the result is the default SBCS CCSID at the current server.

Floating-point to CLOB

floating-point expression
An expression that returns a value that is a built-in floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma.

If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length character string representation of the argument in the form of a floating-point constant.

The length attribute of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0E0.

The CCSID of the result is the default SBCS CCSID at the current server.

Decimal floating-point to CLOB

decimal floating-point expression
An expression that returns a value that is a built-in decimal floating-point data type.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length character string representation of the argument in the form of a decimal floating-point constant.

The length attribute of the result is 42. The actual length of the result is the smallest number of characters that represents the value of the argument, including the sign, digits, and *decimal-character*. Trailing zeros are significant. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0.

If the DECFLOAT value is Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, a minus sign will be the first character in the string. The DECFLOAT special value sNaN does not result in an exception when converted to a string.

The CCSID of the result is the default SBCS CCSID at the current server.

Character to CLOB

character-expression
An expression that returns a value that is a built-in character-string data type.

length
An integer constant that specifies the length attribute for the resulting varying length character string. The value must be between 1 and 2 147 483 647. If the first argument is mixed data, the second argument cannot be less than 4.

If the second argument is not specified or DEFAULT is specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID of the result. It must be a valid SBCS CCSID or mixed data CCSID. If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. If the third argument is a SBCS CCSID, then the first argument cannot be a DBCS-either or DBCS-only string. The third argument cannot be 65535.

If the third argument is not specified, the first argument must not have a CCSID of 65535:

- If the first argument is bit data, an error is returned.
- If the first argument is SBCS data, then the result is SBCS data. The CCSID of the result is the same as the CCSID of the first argument.
- If the first argument is mixed data (DBCS-open, DBCS-only, or DBCS-either), then the result is mixed data. The CCSID of the result is the same as the CCSID of the first argument.

Graphic to CLOB

graphic-expression
An expression that returns a value that is a built-in graphic-string data type. It must not be DBCS-graphic data.

length
An integer constant that specifies the length attribute for the resulting varying length character string. The value must be between 1 and 2 147 483 647. If the result is mixed data, the second argument cannot be less than 4.

If the second argument is not specified or DEFAULT is specified, the length attribute of the result is determined as follows (where *n* is the length attribute of the first argument):

- If the *graphic-expression* is the empty graphic string constant, the length attribute of the result is 1.
- If the result is SBCS data, the result length is *n*.
- If the result is mixed data, the result length is (2.5*(*n*-1)) + 4.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID of the result. It must be a valid SBCS CCSID or mixed data CCSID. If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. The third argument cannot be 65535.

If the third argument is not specified, the CCSID of the result is the default CCSID at the current server. If the default CCSID is mixed data, then the result is mixed data. If the default CCSID is SBCS data, then the result is SBCS data.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the first argument is a string and the length attribute is specified. For more information, see “CAST specification” on page 185.

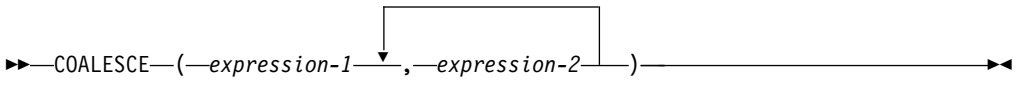
Example

- The following function returns a CLOB for the string 'This is a CLOB'.

```
SELECT CLOB('This is a CLOB')
  FROM SYSIBM.SYSDUMMY1
```

COALESCE

The COALESCE function returns the value of the first non-null expression.



The arguments must be compatible. Character-string arguments are compatible with datetime values. For more information about data type compatibility, see "Assignments and comparisons" on page 98.

expression-1
An expression that returns a value of any built-in or user-defined data type. [52]

expression-2
An expression that returns a value of any built-in or user-defined data type. [52]

The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not null. The result can be null only if all arguments can be null, and the result is null only if all arguments are null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands as explained in "Rules for result data types" on page 115.

Examples

- When selecting all the values from all the rows in the DEPARTMENT table, if the department manager (MGRNO) is missing (that is, null), then return a value of 'ABSENT'.

```
SELECT DEPTNO, DEPTNAME, COALESCE(MGRNO, 'ABSENT'), ADMRDEPT
  FROM DEPARTMENT
```

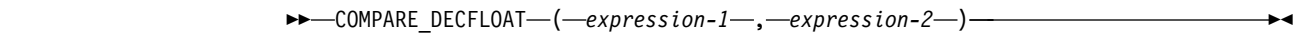
- When selecting the employee number (EMPNO) and salary (SALARY) from all the rows in the EMPLOYEE table, if the salary is missing (that is null), then return a value of zero.

```
SELECT EMPNO, COALESCE(SALARY,0)
  FROM EMPLOYEE
```

52. This function cannot be used as a source function when creating a user-defined function. Because it accepts any compatible data types as arguments, it is not necessary to create additional signatures to support distinct types.

COMPARE_DECFLOAT

The COMPARE_DECFLOAT function returns an ordering for DECFLOAT values.



The COMPARE_DECFLOAT function returns a small integer value that indicates how *expression-1* compares with *expression-2*.

expression-1
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

expression-2
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

The first argument is compared with the second argument and the result is returned according to the following rules.

- If both operands are finite, the comparison is algebraic and follows the rules for DECFLOAT subtraction. If the difference is exactly zero with either sign and with the same number of zeroes to the right of the decimal point, the arguments are equal. If a nonzero difference is positive, the first argument is greater than the second argument. If a nonzero difference is negative, the first argument is less than the second.
- Positive zero and negative zero compare equal.
- Positive Infinity compares equal to positive infinity.
- Positive Infinity compares greater than any finite number.
- Negative Infinity compares equal to negative infinity.
- Negative Infinity compares less than any finite number.
- Numeric comparison is exact. The result is determined for finite operands as if range and precision were unlimited. Overflow or underflow cannot occur.
- If either argument is a NaN or sNaN (positive or negative), the result is unordered.

The result value is set as follows:

0	if the arguments are exactly equal.
1	if *expression-1* is less than *expression-2*.
2	if *expression-1* is greater than *expression-2*.
3	if the arguments are unordered.

The result of the function is SMALLINT. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples

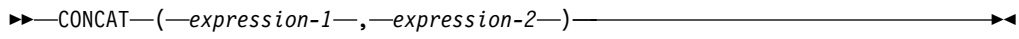
The following examples demonstrate the values that will be returned when the function is used:

```
COMPARE_DECFLOAT (DECFLOAT(2.17),  DECFLOAT(2.17))   =  0
COMPARE_DECFLOAT (DECFLOAT(2.17),  DECFLOAT(2.170))  =  2
COMPARE_DECFLOAT (DECFLOAT(2.170), DECFLOAT(2.17))   =  1
```

```
COMPARE_DECFLOAT (DECFLOAT(2.17),  DECFLOAT(0.0))    =  2
COMPARE_DECFLOAT (INFINITY,        INFINITY)         =  0
COMPARE_DECFLOAT (INFINITY,        -INFINITY)        =  2
COMPARE_DECFLOAT (DECFLOAT(-2),    INFINITY)         =  1
COMPARE_DECFLOAT (NAN,             NAN)              =  3
COMPARE_DECFLOAT (DECFLOAT(-0.1),  SNAN)             =  3
```

CONCAT

The CONCAT function combines two arguments.



The arguments must be compatible. Character-string arguments are not compatible with datetime values. For more information about data type compatibility, see "Assignments and comparisons" on page 98.

expression-1
An expression that returns a value of any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

expression-2
An expression that returns a value of any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The result of the function is a string that consists of the first argument string followed by the second. The data type of the result is determined by the data types of the arguments. For more information, see "With the concatenation operator" on page 170. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Note

Syntax alternatives: The CONCAT function is identical to the CONCAT operator. For more information, see "With the concatenation operator" on page 170.

Example

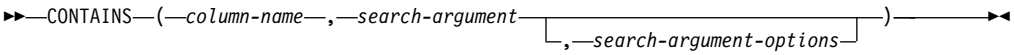
- Concatenate the column FIRSTNME with the column LASTNAME.

```
SELECT CONCAT(FIRSTNME, LASTNAME)
  FROM EMPLOYEE
  WHERE EMPNO ='000010'
```

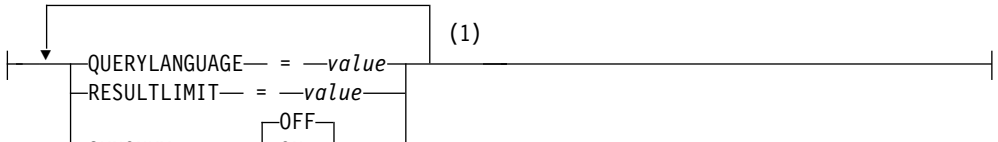
  Returns the value 'CHRISTINEHAAS'.

CONTAINS

The CONTAINS function searches a text search index using criteria that are specified in a search argument and returns a result about whether or not a match was found.



search-argument-options:



Notes:

1 The same clause must not be specified more than once.

column-name
Specifies a qualified or unqualified name of a column that has a text search index that is to be searched. The column must exist in the table or view that is identified in the FROM clause in the statement and the column of the table, or the column of the underlying base table of the view must have an associated text search index. The underlying expression of the column of a view must be a simple column reference to the column of an underlying table, directly or through another nested view.

search-argument
An expression that returns a character-string data type or graphic-string data type that contains the terms to be searched for. It must not be the empty string or contain all blanks. The actual length of the string must not exceed 32 740 bytes after conversion to Unicode and must not exceed the text search limitations or number of terms as specified in the search argument syntax. For information on *search-argument* syntax, see Appendix G, "Text search argument syntax," on page 1805.

search-argument-options
A character string or graphic string value that contains the search argument options to use for the search. It must be a constant or a variable.

The options that can be specified as part of the *search-argument-options* are:

QUERYLANGUAGE = *value*
Specifies the language value. The value can be any of the supported language codes. If QUERYLANGUAGE is not specified, the default is the language value of the text search index that is used when the function is invoked. If the language value of the text search index is AUTO, the default value for QUERYLANGUAGE is en_US. For more information on the query language option, see "Text search language options" on page 1814.

RESULTLIMIT = *value*
Specifies the maximum number of results that are to be returned from the

underlying search engine. The *value* must be an integer from 1 to 2 147 483 647. If RESULTLIMIT is not specified, no result limit is in effect for the query.

CONTAINS may or may not be called for each row of the result table, depending on the plan that the optimizer chooses. If CONTAINS is called once for the query to the underlying search engine, a result set of all of the ROWIDs or primary keys that match are returned from the search engine. This result set is then joined to the table containing the column to identify the result rows. In this case, the RESULTLIMIT value acts like a FETCH FIRST *n* ROWS ONLY from the underlying text search engine and can be used as an optimization. If CONTAINS is called for each row of the result because the optimizer determines that is the best plan, then the RESULTLIMIT option has no effect.

SYNONYM = OFF or SYNONYNM = ON
Specifies whether to use a synonym dictionary associated with the text search index. The default is OFF.

OFF
Do not use a synonym dictionary.

ON Use the synonym dictionary associated with the text search index.

If *search-argument-options* is the empty string or the null value, the function is evaluated as if *search-argument-options* were not specified.

The result of the function is a large integer. If *search-argument* can be null, the result can be null; if *search-argument* is null, the result is the null value.

The result is 1 if the column contains a match for the search criteria specified by the *search-argument*. Otherwise, the result is 0. If the column contains the null value, the result is 0.

CONTAINS is a non-deterministic function.

Notes

Prerequisites: In order to use the CONTAINS and SCORE functions, OmniFind® Text Search Server for DB2 for i must be installed and started.

Rules: If a view, nested table expression, or common table expression provides a text search column for a CONTAINS or SCORE scalar function and the applicable view, nested table expression, or common table expression has a DISTINCT clause on the outermost SELECT, the SELECT list must contain all the corresponding key fields of the text search index.

If a view, nested table expression, or common table expression provides a text search column for a CONTAINS or SCORE scalar function, the applicable view, nested table expression, or common table expression cannot have a UNION, EXCEPT, or INTERSECT at the outermost SELECT.

If a common table expression provides a text search column for a CONTAINS or SCORE scalar function, the common table expression cannot be subsequently referenced again in the entire query unless that reference does not provide a text search column for a CONTAINS or SCORE scalar function.

CONTAINS and SCORE scalar functions are not allowed if the query specifies:

- a distributed table,
- a table with a read trigger, or
- a logical file built over multiple physical file members.

Examples

- The following statement finds all of the employees who have "COBOL" in their resume. The text search argument is not case-sensitive.

```
SELECT EMPNO
 FROM EMP_RESUME
 WHERE RESUME_FORMAT = 'ascii'
    AND CONTAINS(RESUME, 'cobol') = 1
```

- Find 10 students at random whose online essay contains the phrase "fossil fuel" in Spanish, that is "combustible fósil", to be invited for a radio interview. Since any 10 students can be selected, optimize the query to using RESULTLIMIT to limit the number of results from the search.

```
SELECT FIRSTNME, LASTNAME
  FROM STUDENT_ESSAYS
  WHERE CONTAINS(TERM_PAPER, 'combustible fósil',
                 'QUERYLANGUAGE = es_ES RESULTLIMIT = 10 SYNONYM = ON') = 1
```

- Find the string 'ate' in the COMMENT column. Use a host variable to supply the search argument.

```
char search_arg[100];
...
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT CUSTKEY
    FROM CUSTOMERS
    WHERE CONTAINS(COMMENT, :search_arg) = 1
    ORDER BY CUSTKEY;
strcpy(search_arg, "ate");
EXEC SQL OPEN C1;
```

COS

The COS function returns the cosine of the argument, where the argument is an angle expressed in radians. The COS and ACOS functions are inverse operations.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable COSINE is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT COS(:COSINE)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 0.07.

COSH

The COSH function returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable HCOS is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT COSH(:HCOS)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 2.35.

COT

The COT function returns the cotangent of the argument, where the argument is an angle expressed in radians.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable COTAN is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT COT(:COTAN)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 0.07.

CURDATE

The CURDATE function returns a date based on a reading of the time-of-day clock when the SQL statement is executed at the current server. The value returned by the CURDATE function is the same as the value returned by the CURRENT DATE special register.

The data type of the result is a date. The result cannot be null.

If this function is used more than once within a single SQL statement, or used with the CURTIME or NOW scalar functions or the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP special registers within a single statement, all values are based on a single clock reading.

Note

Syntax alternatives: The CURRENT_DATE special register should be used for maximal portability. For more information, see "Special registers" on page 129.

Example

- Return the current date based on the time-of-day clock.

```
SELECT CURDATE()
  FROM SYSIBM.SYSDUMMY1
```

CURTIME

The CURTIME function returns a time based on a reading of the time-of-day clock when the SQL statement is executed at the current server. The value returned by the CURTIME function is the same as the value returned by the CURRENT TIME special register.

The data type of the result is a time. The result cannot be null.

If this function is used more than once within a single SQL statement, or used with the CURDATE or NOW scalar functions or the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP special registers within a single statement, all values are based on a single clock reading.

Note

Syntax alternatives: The CURRENT_TIME special register should be used for maximal portability. For more information, see "Special registers" on page 129.

Example

- Return the current time based on the time-of-day clock.

```
SELECT CURTIME()
  FROM SYSIBM.SYSDUMMY1
```

DATABASE

The DATABASE function returns the current server.

The result of the function is a VARCHAR(18). The result cannot be null.

The CCSID of the string is the default SBCS CCSID at the current server.

Note

Syntax alternatives: The DATABASE function returns the same result as the CURRENT SERVER special register.

Examples

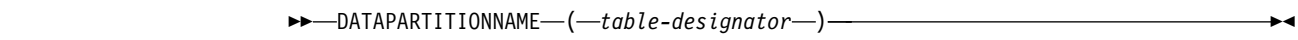
- Assume that the current server is 'RCHASGMA'.

```
SELECT DATABASE( )
  FROM SYSIBM.SYSDUMMY1
```

  Results in a value of ‘RCHASGMA'.

DATAPARTITIONNAME

The DATAPARTITIONNAME function returns the partition name of where a row is located. If the argument identifies a non-partitioned table, an empty string is returned.



For more information about partitions, see the DB2 Multisystem topic collection.

table-designator
A table designator of the subselect. For more information about table designators, see "Table designators" on page 143.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

If the argument identifies a view, common table expression, or nested table expression, the function returns the partition name of its base table. If the argument identifies a view, common table expression, or nested table expression derived from more than one base table, the function returns the partition name of the first table in the outer subselect of the view, common table expression, or nested table expression.

The argument must not identify a view, common table expression, or nested table expression whose outer fullselect subselect includes an aggregate function, a GROUP BY clause, a HAVING clause, a UNION clause, an INTERSECT clause, DISTINCT clause, or VALUES clause. The DATAPARTITIONNAME function cannot be specified in a SELECT clause if the fullselect contains an aggregate function, a GROUP BY clause, a HAVING clause, or a VALUES clause. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is VARCHAR(18). The result can be null.

The CCSID of the result is the default CCSID of the current server.

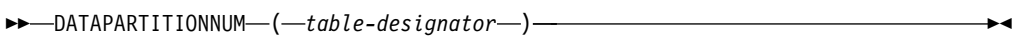
Example

- Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO) and determine the partition from which each row involved in the join originated.

```
SELECT EMPNO, DATAPARTITIONNAME(X), DATAPARTITIONNAME(Y)
  FROM EMPLOYEE X, DEPARTMENT Y
  WHERE X.DEPTNO=Y.DEPTNO
```

DATAPARTITIONNUM

The DATAPARTITIONNUM function returns the data partition number of a row. If the argument identifies a non-partitioned table, the value 0 is returned.



For more information about data partitions, see the DB2 Multisystem topic collection.

table-designator
A table designator of the subselect. For more information about table designators, see "Table designators" on page 143.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

If the argument identifies a view, common table expression, or nested table expression, the function returns the data partition number of its base table. If the argument identifies a view, common table expression, or nested table expression derived from more than one base table, the function returns the data partition number of the first table in the outer subselect of the view, common table expression, or nested table expression.

The argument must not identify a view, common table expression, or nested table expression whose outer fullselect subselect includes an aggregate function, a GROUP BY clause, a HAVING clause, a UNION clause, an INTERSECT clause, DISTINCT clause, or VALUES clause. The DATAPARTITIONNUM function cannot be specified in a SELECT clause if the fullselect contains an aggregate function, a GROUP BY clause, a HAVING clause, or a VALUES clause. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is a large integer. The result can be null.

Example

- Determine the partition number and employee name for each row in the EMPLOYEE table. If this is a partitioned table, the number of the partition where the row exists is returned.

```
SELECT DATAPARTITIONNUM(EMPLOYEE), LASTNAME
  FROM EMPLOYEE
```

DATE

The DATE function returns a date from a value.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or any numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be one of the following:
  - A valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.
  - A string with an actual length of 7 that represents a valid date in the form *yyyynnn*, where *yyyy* are digits denoting a year, and *nnn* are digits between 001 and 366 denoting a day of that year.
- If *expression* is a number, it must be a positive number less than or equal to 3 652 059.

The result of the function is a date. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a timestamp:

  The result is the date part of the timestamp.
- If the argument is a date:

  The result is that date.
- If the argument is a number:

  The result is the date that is *n*-1 days after January 1, 0001, where *n* is the integral part of the number.
- If the argument is a character or graphic string:

  The result is the date represented by the string or the date part of the timestamp value represented by the string.

  When a string representation of a date is SBCS data with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a date value.

  When a string representation of a date is mixed data with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a date value.

  When a string representation of a date is graphic data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a date value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the argument is a date, timestamp, or character string. For more information, see "CAST specification" on page 185.

Examples

- Assume that the column RECEIVED (TIMESTAMP) has an internal value equivalent to '1988-12-25-17.12.30.000000'.

```
SELECT DATE(RECEIVED)
  FROM IN_TRAY
  WHERE SOURCE = 'BADAMSON'
```

  Results in a date data type with a value of '1988-12-25'.

- The following DATE scalar function applied to an ISO string representation of a date:

```
SELECT DATE('1988-12-25')
  FROM SYSIBM.SYSDUMMY1
```

  Results in a date data type with a value of '1988-12-25'.

- The following DATE scalar function applied to an EUR string representation of a date:

```
SELECT DATE('25.12.1988')
  FROM SYSIBM.SYSDUMMY1
```

  Results in a date data type with a value of '1988-12-25'.

- The following DATE scalar function applied to a positive number:

```
SELECT DATE(35)
  FROM SYSIBM.SYSDUMMY1
```

  Results in a date data type with a value of '0001-02-04'.

DAY

The DAY function returns the day part of a value.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.
- If *expression* is a number, it must be a date duration or timestamp duration. For the valid formats of datetime durations, see "Datetime operands and durations" on page 173.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a date, timestamp, or valid character-string representation of a date or timestamp:

  The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a date duration or timestamp duration:

  The result is the day part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Examples

- Using the PROJECT table, set the host variable END_DAY (SMALLINT) to the day that the WELD LINE PLANNING project (PROJNAME) is scheduled to stop (PRENDATE).

```
SELECT DAY(PRENDATE)
  INTO :END_DAY
  FROM PROJECT
  WHERE PROJNAME = 'WELD LINE PLANNING'
```

  Results in END_DAY being set to 15.
- Return the day part of the difference between two dates:

```
SELECT DAY( DATE('2000-03-15') - DATE('1999-12-31') )
  FROM SYSIBM.SYSDUMMY1
```

  Results in the value 15.

DAYNAME

Returns a mixed case character string containing the name of the day (for example, Friday) for the day portion of the argument.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is VARCHAR(100). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the default CCSID of the current server.

Note

National language considerations: The name of the day that is returned is based on the language used for messages in the job. This name of the day is retrieved from message CPX9034 in message file QCPFMSG in library *LIBL.

Examples

- Assume that the language used is US English.

```
SELECT DAYNAME( '2003-01-02' )
  FROM SYSIBM.SYSDUMMY1
```

  Results in 'Thursday'.

DAYOFMONTH

The DAYOFMONTH function returns an integer between 1 and 31 that represents the day of the month.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Using the PROJECT table, set the host variable END_DAY (SMALLINT) to the day that the WELD LINE PLANNING project (PROJNAME) is scheduled to stop (PRENDATE).

```
SELECT DAYOFMONTH(PRENDATE)
  INTO :END_DAY
  FROM PROJECT
  WHERE PROJNAME = 'WELD LINE PLANNING'
```

  Results in END_DAY being set to 15.

DAYOFWEEK

The DAYOFWEEK function returns an integer between 1 and 7 that represents the day of the week, where 1 is Sunday and 7 is Saturday.

For another alternative, see "DAYOFWEEK_ISO" on page 313.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Using the EMPLOYEE table, set the host variable DAY_OF_WEEK (INTEGER) to the day of the week that Christine Haas (EMPNO='000010') started (HIREDATE).

```
SELECT DAYOFWEEK(HIREDATE)
  INTO :DAY_OF_WEEK
  FROM EMPLOYEE
  WHERE EMPNO = '000010'
```

  Results in DAY_OF_WEEK being set to 6, which represents Friday.

- The following query returns four values: 1, 2, 1, and 2.

```
SELECT DAYOFWEEK(CAST('10/11/1998' AS DATE)),
       DAYOFWEEK(TIMESTAMP('10/12/1998','01.02')),
       DAYOFWEEK(CAST(CAST('10/11/1998' AS DATE) AS CHAR(20))),
       DAYOFWEEK(CAST(TIMESTAMP('10/12/1998','01.02') AS CHAR(26)))
  FROM SYSIBM.SYSDUMMY1
```

DAYOFWEEK_ISO

The DAYOFWEEK_ISO function returns an integer between 1 and 7 that represents the day of the week, where 1 is Monday and 7 is Sunday.

For another alternative, see "DAYOFWEEK_ISO."

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Using the EMPLOYEE table, set the host variable DAY_OF_WEEK (INTEGER) to the day of the week that Christine Haas (EMPNO='000010') started (HIREDATE).

```
SELECT DAYOFWEEK_ISO(HIREDATE)
  INTO :DAY_OF_WEEK
  FROM EMPLOYEE
  WHERE EMPNO = '000010'
```

  Results in DAY_OF_WEEK being set to 5, which represents Friday.

- The following query returns four values: 7, 1, 7, and 1.

```
SELECT DAYOFWEEK_ISO(CAST('10/11/1998' AS DATE)),
       DAYOFWEEK_ISO(TIMESTAMP('10/12/1998','01.02')),
       DAYOFWEEK_ISO(CAST(CAST('10/11/1998' AS DATE) AS CHAR(20))),
       DAYOFWEEK_ISO(CAST(TIMESTAMP('10/12/1998','01.02') AS CHAR(26)))
  FROM SYSIBM.SYSDUMMY1
```

DAYOFYEAR

The DAYOFYEAR function returns an integer between 1 and 366 that represents the day of the year where 1 is January 1.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Using the EMPLOYEE table, set the host variable AVG_DAY_OF_YEAR (INTEGER) to the average of the day of the year that employees started on (HIREDATE).

```
SELECT AVG(DAYOFYEAR(HIREDATE))
  INTO :AVG_DAY_OF_YEAR
  FROM EMPLOYEE
```

  Results in AVG_DAY_OF_YEAR being set to 197.

DAYS

The DAYS function returns an integer representation of a date.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is 1 more than the number of days from January 1, 0001 to *D*, where *D* is the date that would occur if the DATE function were applied to the argument.

Examples

- Using the PROJECT table, set the host variable EDUCATION_DAYS (INTEGER) to the number of elapsed days (PRENDATE - PRSTDATE) estimated for the project (PROJNO) 'IF2000'.

```
SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
  INTO :EDUCATION_DAYS
  FROM PROJECT
  WHERE PROJNO = 'IF2000'
```

  Results in EDUCATION_DAYS being set to 396.

- Using the PROJECT table, set the host variable TOTAL_DAYS (INTEGER) to the sum of elapsed days (PRENDATE - PRSTDATE) estimated for all projects in department (DEPTNO) 'E21'.

```
SELECT SUM(DAYS(PRENDATE) - DAYS(PRSTDATE))
  INTO :TOTAL_DAYS
  FROM PROJECT
  WHERE DEPTNO = 'E21'
```

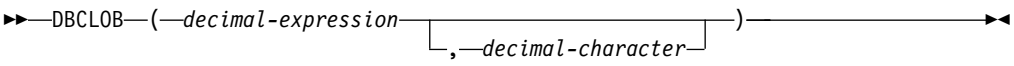
  Results in TOTAL_DAYS being set to 1584.

DBCLOB

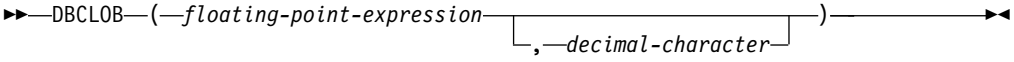
The DBCLOB function returns a graphic-string representation.

Integer to DBCLOB

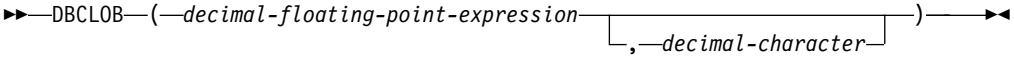
Decimal to DBCLOB



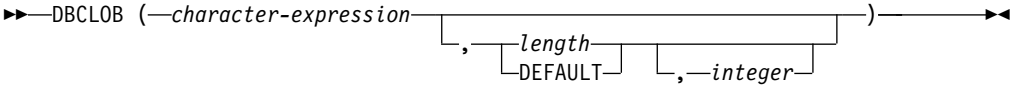
Floating-point to DBCLOB



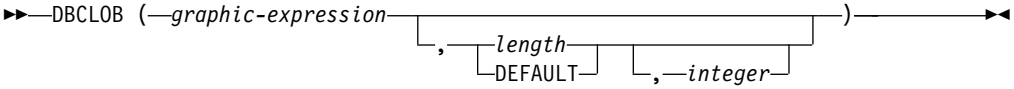
Decimal floating-point to DBCLOB



Character to DBCLOB



Graphic to DBCLOB



The DBCLOB function returns a graphic-string representation of:

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number if the first argument is a packed or zoned decimal number
- A double-precision floating-point number if the first argument is a DOUBLE or REAL
- A decimal floating-point number if the first argument is a DECFLOAT
- A character string if the first argument is any type of character string
- A graphic string if the first argument is any type of graphic string

The result of the function is a DBCLOB. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Integer to DBCLOB

integer-expression
An expression that returns a value that is a built-in integer data type (either SMALLINT, INTEGER, or BIGINT).

The result is a varying-length graphic string of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. The result is left justified.

- If the argument is a small integer, the length attribute of the result is 6.
- If the argument is a large integer, the length attribute of the result is 11.
- If the argument is a big integer, the length attribute of the result is 20.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit or the *decimal-character*.

The CCSID of the result is 1200 (UTF-16).

Decimal to DBCLOB

decimal-expression
An expression that returns a value that is a built-in decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is wanted, the DECIMAL scalar function can be used to make the change.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length graphic string representation of the argument. The result includes a decimal character and up to *p* digits, where *p* is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length attribute of the result is 2+*p* where *p* is the precision of the *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing characters are included. Leading zeros are not included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit or the *decimal-character*.

The CCSID of the result is 1200 (UTF-16).

Floating-point to DBCLOB

floating-point expression
An expression that returns a value that is a built-in floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma.

If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length graphic string representation of the argument in the form of a floating-point constant.

The length attribute of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0E0.

The CCSID of the result is 1200 (UTF-16).

Decimal floating-point to DBCLOB

decimal floating-point expression
An expression that returns a value that is a built-in decimal floating-point data type.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length graphic string representation of the argument in the form of a decimal floating-point constant.

The length attribute of the result is 42. The actual length of the result is the smallest number of characters that represents the value of the argument, including the sign, digits, and *decimal-character*. Trailing zeros are significant. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0.

If the DECFLOAT value is Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, a minus sign will be the first character in the string. The DECFLOAT special value sNaN does not result in an exception when converted to a string.

The CCSID of the result is 1200 (UTF-16).

Character to DBCLOB

character-expression
An expression that returns a value that is a built-in character-string data type. It cannot be CHAR or VARCHAR bit data. If the expression is an empty string or the EBCDIC string `X'0E0F'`, the result is an empty string.

length
An integer constant that specifies the length attribute for the resulting varying length character string. The value must be between 1 and 1 073 741 823.

If the second argument is not specified or DEFAULT is specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 1.

- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID for the resulting varying-length graphic string. It must be a DBCS, UTF-16, or UCS-2 CCSID. The CCSID cannot be 65535.

In the following rules, S denotes one of the following:

- If the string expression is a host variable containing data in a foreign encoding scheme, S is the result of the expression after converting the data to a CCSID in a native encoding scheme. (See "Character conversion" on page 32 for more information.)
- If the string expression is data in a native encoding scheme, S is that string expression.

If the third argument is not specified and the first argument is character, then the CCSID of the result is determined by a mixed CCSID. Let M denote that mixed CCSID. M is determined as follows:

- If the CCSID of S is a mixed CCSID, M is that CCSID.
- If the CCSID of S is an SBCS CCSID:
  - If the CCSID of S has an associated mixed CCSID, M is that CCSID.
  - Otherwise the operation is not allowed.

The following table summarizes the result CCSID based on M.

M	Result CCSID	Description	DBCS Substitution Character
930	300	Japanese EBCDIC	X'FEFE'
933	834	Korean EBCDIC	X'FEFE'
935	837	S-Chinese EBCDIC	X'FEFE'
937	835	T-Chinese EBCDIC	X'FEFE'
939	300	Japanese EBCDIC	X'FEFE'
5026	4396	Japanese EBCDIC	X'FEFE'
5035	4396	Japanese EBCDIC	X'FEFE'

If the result is DBCS-graphic data, the equivalence of SBCS and DBCS characters depends on M. Regardless of the CCSID, every double-byte code point in the argument is considered a DBCS character, and every single-byte code point in the argument is considered an SBCS character with the exception of the EBCDIC mixed data shift codes `X'0E'` and `X'0F'`.

- If the nth character of the argument is a DBCS character, the nth character of the result is that DBCS character.
- If the nth character of the argument is an SBCS character that has an equivalent DBCS character, the nth character of the result is that equivalent DBCS character.
- If the nth character of the argument is an SBCS character that does not have an equivalent DBCS character, the nth character of the result is the DBCS substitution character.

If the result is Unicode graphic data, each character of the argument determines a character of the result. The nth character of the result is the UTF-16 or UCS-2 equivalent of the nth character of the argument.

Graphic to DBCLOB

graphic-expression
An expression that returns a value that is a built-in graphic-string data type.

length
An integer constant that specifies the length attribute for the resulting varying length character string. The value must be between 1 and 1 073 741 823.

If the second argument is not specified or DEFAULT is specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID for the resulting varying-length graphic string. It must be a DBCS, UTF-16, or UCS-2 CCSID. The CCSID cannot be 65535.

In the following rules, S denotes one of the following:

- If the string expression is a host variable containing data in a foreign encoding scheme, S is the result of the expression after converting the data to a CCSID in a native encoding scheme. (See “Character conversion” on page 32 for more information.)
- If the string expression is data in a native encoding scheme, S is that string expression.

If the third argument is not specified, then the CCSID of the result is the same as the CCSID of the first argument.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the first argument is a string and the length attribute is specified. For more information, see “CAST specification” on page 185.

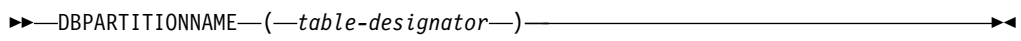
Example

- Using the EMPLOYEE table, set the host variable VAR_DESC (VARGRAPHIC(24)) to the DBCLOB equivalent of the first name (FIRSTNME) for employee number (EMPNO) '000050'.

```
SELECT DBCLOB(VARGRAPHIC(FIRSTNME))
  INTO :VAR_DESC
  FROM EMPLOYEE
  WHERE EMPNO = '000050'
```

DBPARTITIONNAME

The DBPARTITIONNAME function returns the relational database name (database partition name) of where a row is located. If the argument identifies a non-distributed table, the current server is returned.



For more information about partitions, see the DB2 Multisystem topic collection.

table-designator
A table designator of the subselect. For more information about table designators, see "Table designators" on page 143.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

If the argument identifies a view, common table expression, or nested table expression, the function returns the relational database name of its base table. If the argument identifies a view, common table expression, or nested table expression derived from more than one base table, the function returns the partition name of the first table in the outer subselect of the view, common table expression, or nested table expression.

The argument must not identify a view, common table expression, or nested table expression whose outer fullselect subselect includes an aggregate function, a GROUP BY clause, a HAVING clause, a UNION clause, an INTERSECT clause, DISTINCT clause, or VALUES clause. The DBPARTITIONNAME function cannot be specified in a SELECT clause if the fullselect contains an aggregate function, a GROUP BY clause, a HAVING clause, or a VALUES clause. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is VARCHAR(18). The result can be null.

The CCSID of the result is the default CCSID of the current server.

Note

Syntax alternatives: NODENAME is a synonym for DBPARTITIONNAME.

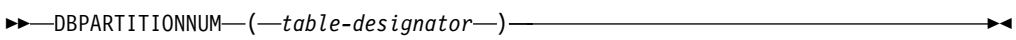
Example

- Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO) and determine the node from which each row involved in the join originated.

```
SELECT EMPNO, DBPARTITIONNAME(X), DBPARTITIONNAME(Y)
  FROM EMPLOYEE X, DEPARTMENT Y
  WHERE X.DEPTNO=Y.DEPTNO
```

DBPARTITIONNUM

The DBPARTITIONNUM function returns the node number (database partition number) of a row.



If the argument identifies a non-distributed table, the value 0 is returned.[53] For more information about nodes and node numbers, see the DB2 Multisystem book.

table-designator
A table designator of the subselect. For more information about table designators, see "Table designators" on page 143.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

If the argument identifies a view, common table expression, or nested table expression, the function returns the node number of its base table. If the argument identifies a view, common table expression, or nested table expression derived from more than one base table, the function returns the node number of the first table in the outer subselect of the view, common table expression, or nested table expression.

The argument must not identify a view, common table expression, or nested table expression whose outer fullselect subselect includes an aggregate function, a GROUP BY clause, a HAVING clause, a UNION clause, an INTERSECT clause, DISTINCT clause, or VALUES clause. The DBPARTITIONNUM function cannot be specified in a SELECT clause if the fullselect contains an aggregate function, a GROUP BY clause, a HAVING clause, or a VALUES clause. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is a large integer. The result can be null.

Note

Syntax alternatives: NODENUMBER is a synonym for DBPARTITIONNUM.

Example

- Determine the node number and employee name for each row in the EMPLOYEE table. If this is a distributed table, the number of the node where the row exists is returned.

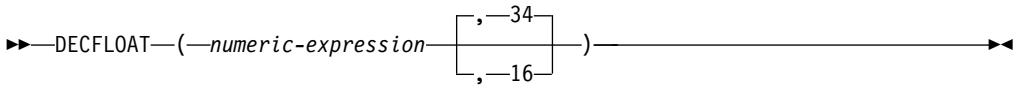
```
SELECT DBPARTITIONNUM(EMPLOYEE), LASTNAME
  FROM EMPLOYEE
```

53. If the argument identifies a DDS created logical file that is based on more than one physical file member, DBPARTITIONNUM will not return 0, but instead will return the underlying physical file member number.

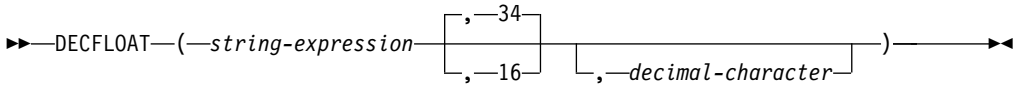
DECFLOAT

The DECFLOAT function returns a decimal floating-point representation of a number or a string representation of a number.

Numeric to DECFLOAT



String to DECFLOAT



The DECFLOAT function returns a decimal floating-point representation of:

- A number
- A character or graphic string representation of a decimal number
- A character or graphic string representation of an integer
- A character or graphic string representation of a floating-point number
- A character or graphic string representation of a decimal floating-point number

Numeric to DECFLOAT

numeric-expression
An expression that returns a value of any built-in numeric data type.

34 **or** *16*
Specifies the number of digits of precision for the result. The default is 34.

The result is the same number that would occur if the first argument were assigned to a decimal floating-point column or variable.

String to DECFLOAT

string-expression
An expression that returns a value that is a character-string or graphic-string representation of a number. Leading and trailing blanks are eliminated and the resulting string folded to uppercase must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant.

34 **or** *16*
Specifies the number of digits of precision for the result. The default is 34.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in *string-expression* from the whole part of the number. The character must be a period or comma. If *decimal-character* is not specified, the decimal point is the default decimal separator character. For more information, see "Decimal point" on page 127.

The result is the same number that would result from CAST(*string-expression* AS DECFLOAT(34)) or CAST(*string-expression* AS DECFLOAT(16)).

The result of the function is a DECFLOAT number with the specified (either implicitly or explicitly) number of digits of precision. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

If necessary, the source is rounded to the precision of the target. See "CURRENT DECFLOAT ROUNDING MODE" on page 133 for more information.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

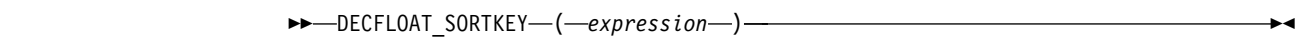
Example

- Use the DECFLOAT function in order to force a DECFLOAT data type to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```
SELECT EMPNO, DECFLOAT(EDLEVEL,16)
  FROM EMPLOYEE
```

DECFLOAT_SORTKEY

The DECFLOAT_SORTKEY function returns a binary value that may be used to sort DECFLOAT values.



The DECFLOAT_SORTKEY function returns a binary value that may be used to sort decimal floating-point values in a manner that is consistent with the IEEE 754R specification on total ordering.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type.

If the data type of the argument is SMALLINT, INTEGER, REAL, DOUBLE, DECIMAL(p,s) where p <=16, or NUMERIC(p,s) where p <=16, then the argument is converted to DECFLOAT(16) for processing. Otherwise, the argument is converted to DECFLOAT(34) for processing.

The result of the function is BINARY(9) if the argument is DECFLOAT(16) or BINARY(17) if the argument is DECFLOAT(34).

If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example

```
CREATE TABLE T1 (D1 DECFLOAT(16));
INSERT INTO T1 VALUES(2.100);
INSERT INTO T1 VALUES(2.10);
INSERT INTO T1 VALUES(2.1000);
INSERT INTO T1 VALUES(2.1);

SELECT D1 FROM T1 ORDER BY D1;

     D1
---------------
          2.100
           2.10
         2.1000
            2.1
```

Note that this result set is arbitrary. The ORDER BY has no effect on ordering these values.

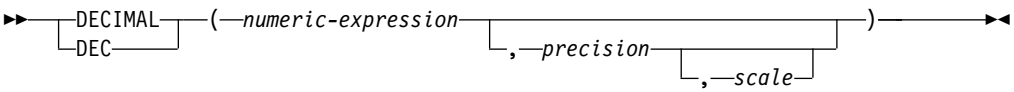
```
SELECT D1 FROM T1 ORDER BY DECFLOAT_SORTKEY(D1);

     D1
---------------
         2.1000
          2.100
           2.10
            2.1
```

Note that this result set is ordered according to the IEEE 745R ordering specification.

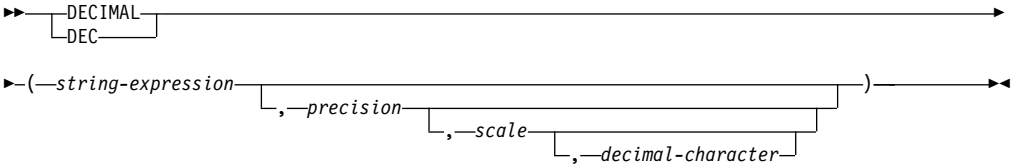
DECIMAL or DEC

The DECIMAL function returns a decimal representation.

Numeric to Decimal



String to Decimal



The DECIMAL function returns a decimal representation of:

- A number
- A character or graphic string representation of a decimal number
- A character or graphic string representation of an integer
- A character or graphic string representation of a floating-point number
- A character or graphic string representation of a decimal floating-point number

Numeric to Decimal

numeric-expression
An expression that returns a value of any built-in numeric data type.

precision
An integer constant with a value greater than or equal to 1 and less than or equal to 63.

The default for *precision* depends on the data type of the *numeric-expression*:

- 5 for small integer
- 11 for large integer
- 19 for big integer
- 15 for floating point, decimal, numeric, or nonzero scale binary
- 31 for decimal floating point

scale
An integer constant that is greater than or equal to 0 and less than or equal to *precision*. If not specified, the default is 0.

The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of *precision* and a scale of *scale*. An error is returned if the number of significant decimal digits required to represent the whole part of the number is greater than *precision-scale*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

String to Decimal

string-expression
An expression that returns a character-string or graphic-string representation of a number. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant.

precision
An integer constant that is greater than or equal to 1 and less than or equal to 63. If not specified, the default is 15.

scale
An integer constant that is greater than or equal to 0 and less than or equal to *precision*. If not specified, the default is 0.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in *string-expression* from the whole part of the number. The character must be a period or comma. If *decimal-character* is not specified, the decimal point is the default decimal separator character. For more information, see "Decimal point" on page 127.

The result is the same number that would result from CAST(*string-expression* AS DECIMAL(*precision*,*scale*)). Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than *scale*. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in *string-expression* is greater than *precision-scale*. The default decimal character is not valid in the substring if the *decimal-character* argument is specified.

The result of the function is a decimal number with precision of *precision* and scale of *scale*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when precision is specified. For more information, see "CAST specification" on page 185.

Examples

- Use the DECIMAL function in order to force a DECIMAL data type (with a precision of 5 and a scale of 2) to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```
SELECT EMPNO, DECIMAL(EDLEVEL,5,2)
  FROM EMPLOYEE
```

- Using the PROJECT table, select all of the starting dates (PRSTDATE) that have been incremented by a duration that is specified in a host variable. Assume the host variable PERIOD is of type INTEGER. Then, in order to use its value as a date duration it must be "cast" as DECIMAL(8,0).

```
SELECT PRSTDATE + DECIMAL(:PERIOD,8)
  FROM PROJECT
```

- Assume that updates to the SALARY column are input through a window as a character string using comma as a decimal character (for example, the user

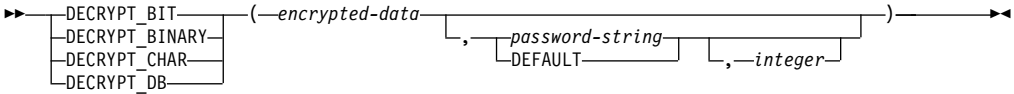
inputs 21400,50). Once validated by the application, it is assigned to the host variable newsalary which is defined as CHAR(10).

```
UPDATE STAFF
  SET SALARY = DECIMAL(:newsalary, 9, 2, ',')
  WHERE ID = :empid
```

The value of SALARY becomes 21400.50.

DECRYPT_BIT, DECRYPT_BINARY, DECRYPT_CHAR and DECRYPT_DB

The DECRYPT_BIT, DECRYPT_BINARY, DECRYPT_CHAR, and DECRYPT_DB functions return a value that is the result of decrypting encrypted data. The password used for decryption is either the *password-string* value or the ENCRYPTION PASSWORD value assigned by the SET ENCRYPTION PASSWORD statement.



The decryption functions can only decrypt values that are encrypted using the ENCRYPT_AES, ENCRYPT_RC2, or ENCRYPT_TDES function.

encrypted-data
An expression that must be a string expression that returns a complete, encrypted data value of a CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, VARBINARY, or BLOB built-in data type. The data string must have been encrypted using the ENCRYPT_AES, ENCRYPT_RC2, or ENCRYPT_TDES function.

password-string
An expression that returns a character string value with at least 6 bytes and no more than 127 bytes. The expression must not be a CLOB. This expression must be the same password used to encrypt the data or decryption will result in a different value than was originally encrypted. If the value of the password argument is null or not provided, the data will be decrypted using the ENCRYPTION PASSWORD value, which must have been set using the SET ENCRYPTION PASSWORD statement.

DEFAULT
The data will be decrypted using the ENCRYPTION PASSWORD value, which must have been set using the SET ENCRYPTION PASSWORD statement.

integer
An integer constant that specifies the CCSID of the result. If DECRYPT_BIT or DECRYPT_BINARY is specified, the third argument must not be specified.

If DECRYPT_CHAR is specified, *integer* must be a valid SBCS CCSID or mixed data CCSID. It cannot be 65535 (bit data). If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. If the third argument is not specified then the CCSID of the result is the default CCSID of the current server.

If DECRYPT_DB is specified, *integer* must be a valid DBCS CCSID. If the third argument is not specified then the CCSID of the result is the DBCS CCSID associated with the default CCSID of the current server.

The data type of the result is determined by the function specified and the data type of the first argument as shown in the following table. If a cast from the actual type of the encrypted data to the function's result is not supported a warning or error is returned.

Function	Data Type of First Argument	Actual Data Type of Encrypted Data	Result
DECRYPT_BIT	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Character string	VARCHAR FOR BIT DATA
DECRYPT_BIT	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Graphic string	Error or Warning **
DECRYPT_BIT	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Binary string	Error or Warning **
DECRYPT_BIT	BLOB	Any string	Error
DECRYPT_BINARY	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Any string	VARBINARY
DECRYPT_BINARY	BLOB	Any string	BLOB
DECRYPT_CHAR	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Character string	VARCHAR
DECRYPT_CHAR	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Unicode graphic string	VARCHAR
DECRYPT_CHAR	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Non-Unicode graphic string	Error or Warning **
DECRYPT_CHAR	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Binary string	Error or Warning **
DECRYPT_CHAR	BLOB	Character string	CLOB
DECRYPT_CHAR	BLOB	Unicode graphic string	CLOB
DECRYPT_CHAR	BLOB	Non-Unicode graphic string	Error or Warning **
DECRYPT_CHAR	BLOB	Binary string	Error or Warning **
DECRYPT_DB	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	UTF-8 character string or graphic string	VARGRAPHIC
DECRYPT_DB	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Non-UTF-8 character string	Error or Warning **
DECRYPT_DB	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY	Binary string	Error or Warning **
DECRYPT_DB	BLOB	UTF-8 character string or graphic string	DBCLOB
DECRYPT_DB	BLOB	Non-UTF-8 character string	Error or Warning **
DECRYPT_DB	BLOB	Binary string	Error or Warning **

Function	Data Type of First Argument	Actual Data Type of Encrypted Data	Result
Note: ** If the decryption function is in the select list of an outer subselect, a data mapping warning is returned. Otherwise an error is returned. For more information about data mapping warnings, see "Assignments and comparisons" on page 98.			

If the *encrypted-data* included a hint, the hint is not returned by the function. The length attribute of the result is the length attribute of the data type of *encrypted-data* minus 8 bytes. The actual length of the result is the length of the original string that was encrypted. If the *encrypted-data* includes bytes beyond the encrypted string, these bytes are not returned by the function.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the data is decrypted using a different CCSID than the originally encrypted value, expansion may occur when converting the decrypted value to this CCSID. In such situations, the *encrypted-data* should be cast to a varying-length string with a larger number of bytes.

Note

Password protection: To prevent inadvertent access to the encryption password, do not specify *password-string* as a string constant in the source for a program, procedure, or function. Instead, use the ENCRYPTION PASSWORD special register or a host variable.

When connected to a remote relational database, the specified password itself is sent "in the clear". That is, the password itself is not encrypted. To protect the password in these cases, consider using a communications encryption mechanism such as IPSEC (or SSL if connecting between IBM i products).

Syntax alternatives: For compatibility with previous versions of DB2, DECRYPT_BIN can be specified in place of DECRYPT_BIT.

Examples

- Assume that table EMP1 has a social security column called SSN. This example uses the ENCRYPTION PASSWORD value to hold the encryption password.

```
SET ENCRYPTION PASSWORD = :pw

INSERT INTO EMP1 (SSN) VALUES ENCRYPT_RC2( '289-46-8832' )

SELECT DECRYPT_CHAR( SSN)
  FROM EMP1
```

  The DECRYPT_CHAR function returns the original value '289-46-8832'.

- This example explicitly passes the encryption password which has been set in variable pw.

```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT_TDES( '289-46-8832', :pw)

SELECT DECRYPT_CHAR( SSN, :pw)
  FROM EMP1
```

  The DECRYPT_CHAR function returns the original value '289-46-8832'.

DEGREES

The DEGREES function returns the number of degrees of the argument which is an angle expressed in radians.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

If the data type of the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

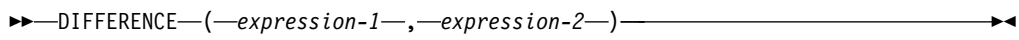
- Assume the host variable RAD is a DECIMAL(4,3) host variable with a value of 3.142.

```
SELECT DEGREES(:RAD)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 180.0.

DIFFERENCE

The DIFFERENCE function returns a value from 0 to 4 representing the difference between the sounds of two strings based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.



expression-1
An expression that returns a built-in numeric, character-string, or graphic-string data types, but not CLOBs or DBCLOBs. The arguments cannot be binary strings. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

expression-2
An expression that returns a built-in numeric, character-string, or graphic-string data types, but not CLOBs or DBCLOBs. The arguments cannot be binary strings. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The data type of the result is INTEGER. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples

- Assume the following statement:

```
SELECT DIFFERENCE('CONSTRAINT','CONSTANT'),
       SOUNDEX('CONSTRAINT'),
       SOUNDEX('CONSTANT')
  FROM SYSIBM.SYSDUMMY1
```

  Returns 4, C523, and C523. Since the two strings return the same SOUNDEX value, the difference is 4 (the highest value possible).

- Assume the following statement:

```
SELECT DIFFERENCE('CONSTRAINT','CONTRITE'),
       SOUNDEX('CONSTRAINT'),
       SOUNDEX('CONTRITE')
  FROM SYSIBM.SYSDUMMY1
```

  Returns 2, C523, and C536. In this case, the two strings return different SOUNDEX values, and hence, a lower difference value.

DIGITS

The DIGITS function returns a character-string representation of the absolute value of a number.

expression
An expression that returns a value of a built-in SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, character-string, or graphic-string data type. A string argument is cast to DECIMAL(63,31) before evaluating the function. For more information about converting strings to decimal, see "DECIMAL or DEC" on page 326.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a fixed-length character string representing the absolute value of the argument without regard to its scale. The result does not include a sign or a decimal point. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer with a scale of zero
- 10 if the argument is a large integer with a scale of zero
- 19 if the argument is a big integer
- *p* if the argument is a decimal (or an integer with a scale greater than zero) with a precision of *p*

The CCSID of the character string is the default SBCS CCSID at the current server.

Examples

- Assume that a table called TABLEX contains an INTEGER column called INTCOL containing 10-digit numbers. List all combinations of the first four digits contained in column INTCOL.

```
SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
  FROM TABLEX
```

- Assume that COLUMNX has the DECIMAL(6,2) data type, and that one of its values is -6.28.

```
SELECT DIGITS(COLUMNX)
  FROM TABLEX
```

  Returns the value '000628'.

  The result is a string of length six (the precision of the column) with leading zeros padding the string out to this length. Neither sign nor decimal point appear in the result.

DLCOMMENT

The DLCOMMENT function returns the comment value, if it exists, from a DataLink value.

DataLink-expression
An expression that results in a value with a built-in DataLink data type.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Prepare a statement to select the date, the description and the comment from the link to the ARTICLES column from the HOCKEY_GOALS table. The rows to be selected are those for goals scored by either of the Richard brothers (Maurice or Henri).

```
stmtvar = "SELECT DATE_OF_GOAL, DESCRIPTION, DLCOMMENT(ARTICLES)
                 FROM HOCKEY_GOALS
                 WHERE BY_PLAYER = 'Maurice Richard'
                 OR BY_PLAYER = 'Henri Richard' ";
EXEC SQL PREPARE HOCKEY_STMT FROM :stmtvar;
```

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TBLA
  VALUES (DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment'))
```

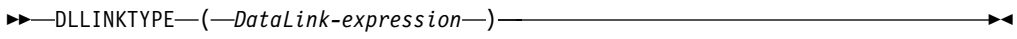
  then the following function operating on that value:

```
SELECT DLCOMMENT(COLA)
  FROM TBLA
```

  Returns the value 'A comment'.

DLLINKTYPE

The DLLINKTYPE function returns the link type value from a DataLink value.



DataLink-expression
An expression that results in a value with a built-in DataLink data type.

The result of the function is VARCHAR(4). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
  VALUES( DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment') )
```

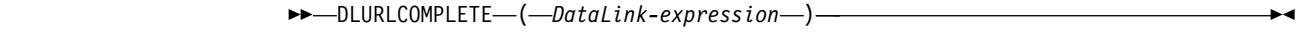
  then the following function operating on that value:

```
SELECT DLLINKTYPE(COLA)
  FROM TBLA
```

  Returns the value 'URL'.

DLURLCOMPLETE

The DLURLCOMPLETE function returns the complete URL value from a DataLink value with a link type of URL. The value is the same as what would be returned by the concatenation of DLURLSCHEME with '://', then DLURLSERVER, and then DLURLPATH. If the DataLink has an attribute of FILE LINK CONTROL and READ PERMISSION DB, the value includes a file access token.



DataLink-expression
An expression that results in a value with a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a varying-length string. The length attribute depends on the attributes of the DataLink:

- If the DataLink has an attribute of FILE LINK CONTROL and READ PERMISSION DB, the length attribute of the result is the length attribute of the argument plus 19.
- Otherwise, the length attribute of the result is the length attribute of the argument.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA (with the attributes of FILE LINK CONTROL and READ PERMISSION DB) of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
  VALUES( DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment') )
```

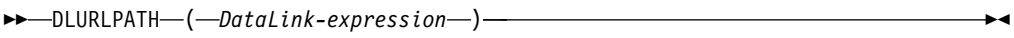
  then the following function operating on that value:

```
SELECT DLURLCOMPLETE(COLA)
  FROM TBLA
```

  Returns the value 'HTTP://DLFS.ALMADEN.IBM.COM/x/y/**********;a.b', where ********** represents the access token.

DLURLPATH

The DLURLPATH function returns the path and file name necessary to access a file within a given server from a DataLink value with a linktype of URL. When appropriate, the value includes a file access token.



DataLink-expression
An expression that results in a value with a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a varying-length string. The length attribute depends on the attributes of the DataLink:

- If the DataLink has an attribute of FILE LINK CONTROL and READ PERMISSION DB, the length attribute of the result is the length attribute of the argument plus 19.
- Otherwise, the length attribute of the result is the length attribute of the argument.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA (with the attributes of FILE LINK CONTROL and READ PERMISSION DB) of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
  VALUES( DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment') )
```

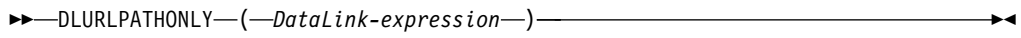
  then the following function operating on that value:

```
SELECT DLURLPATH(COLA)
  FROM TBLA
```

  Returns the value '/x/y/**********;a.b', where ********** represents the access token.

DLURLPATHONLY

The DLURLPATHONLY function returns the path and file name necessary to access a file within a given server from a DataLink value with a linktype of URL. The value returned NEVER includes a file access token.



DataLink-expression
An expression that results in a value with a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a varying-length string with a length attribute of that is equal to the length attribute of the argument.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
  VALUES( DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment') )
```

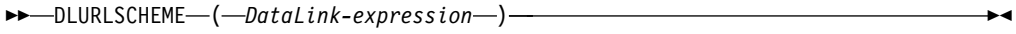
  then the following function operating on that value:

```
SELECT DLURLPATHONLY(COLA)
  FROM TBLA
```

  Returns the value '/x/y/a.b'.

DLURLSCHEME

The DLURLSCHEME function returns the scheme from a DataLink value with a linktype of URL. The value will always be in upper case.



DataLink-expression
An expression that results in a value with a built-in DataLink data type.

The result of the function is VARCHAR(20). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
  VALUES( DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment') )
```

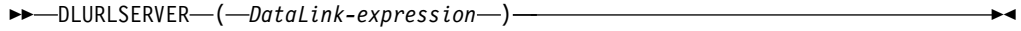
  then the following function operating on that value:

```
SELECT DLURLSCHEME(COLA)
  FROM TBLA
```

  Returns the value 'HTTP'.

DLURLSERVER

The DLURLSERVER function returns the file server from a DataLink value with a linktype of URL. The value will always be in upper case.



DataLink-expression
An expression that results in a value with a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a varying-length string with a length attribute of that is equal to the length attribute of the argument.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
  VALUES( DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment') )
```

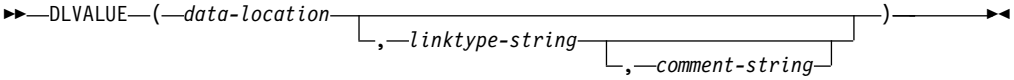
  then the following function operating on that value:

```
SELECT DLURLSERVER(COLA)
  FROM TBLA
```

  Returns the value 'DLFS.ALMADEN.IBM.COM'.

DLVALUE

The DLVALUE function returns a DataLink value. When the function is on the right hand side of a SET clause in an UPDATE statement or is in a VALUES clause in an INSERT statement, it usually also creates a link to a file. However, if only a comment is specified (in which case the *data-location* is a zero-length string), the DataLink value is created with empty linkage attributes so there is no file link.



data-location
If the link type is URL, then this is a character string expression that contains a complete URL value. If the expression is not an empty string, it must include the URL scheme and URL server. The actual length of the character string expression must be less than or equal to 32718 characters.

linktype-string
An optional character string expression that specifies the link type of the DataLink value. The only valid value is 'URL'.

comment-string
An optional character string expression that provides a comment or additional location information. The actual length of the character string expression must be less than or equal to 254 characters.

The *comment-string* cannot be the null value. If a *comment-string* is not specified, the *comment-string* is the empty string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The result of the function is a DataLink value.

The CCSID of the DataLink is the same as that of *data-location* except in the following cases:

- If the *comment string* is mixed data and *data-location* is not mixed data, the CCSID of the result will be the CCSID of the *comment string*.[54]
- If the *data-location* has a CCSID of bit data (65535), UTF-16 graphic data (1200), UCS-2 graphic data (13488), Turkish data (905 or 1026), or Japanese data (290, 930, or 5026); the CCSID of the result is described in the following table:

CCSID of *data-location*	**CCSID of** *comment-string*	**Result CCSID**
65535	65535	Job Default CCSID
65535	non-65535	*comment-string* CCSID (unless the CCSID is 290, 930, 5026, 905, 1026, or 13488 where the CCSID will then be further modified as described in the following rows.)
290	any	4396
930 or 5026	any	939
905 or 1026	any	500

54. If the CCSID of *comment string* is 5026 or 930, the CCSID of the results will be 939.

CCSID of *data-location*	**CCSID of** *comment-string*	**Result CCSID**
1200	any	500
13488	any	500

When defining a DataLink value using this function, consider the maximum length of the target of the value. For example, if a column is defined as DataLink(200), then the maximum length of the *data-location* plus the comment is 200 bytes.

Examples

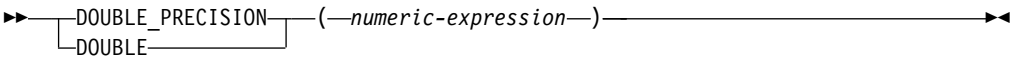
- Insert a row into the table. The URL values for the first two links are contained in the variables named url_article and url_snapshot. The variable named url_snapshot_comment contains a comment to accompany the snapshot link. There is, as yet, no link for the movie, only a comment in the variable named url_movie_comment.

```
INSERT INTO HOCKEY_GOALS
  VALUES('Maurice Richard',
         'Montreal canadian',
         '?',
         'Boston Bruins,
         '1952-04-24',
         'Winning goal in game 7 of Stanley Cup final',
         DLVALUE(:url_article),
         DLVALUE(:url_snapshot, 'URL', :url_snapshot_comment),
         DLVALUE('', 'URL', :url_movie_comment) )
```

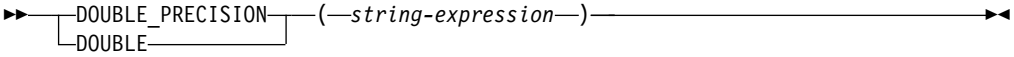
DOUBLE_PRECISION or DOUBLE

The DOUBLE_PRECISION and DOUBLE functions return a floating-point representation.

Numeric to Double



String to Double



The DOUBLE_PRECISION and DOUBLE functions return a floating-point representation of:

- A number
- A character or graphic string representation of a decimal number
- A character or graphic string representation of an integer
- A character or graphic string representation of a floating-point number
- A character or graphic string representation of a decimal floating-point number

Numeric to Double

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the expression were assigned to a double-precision floating-point column or variable.

String to Double

string-expression
An expression that returns a value that is a character-string or graphic-string representation of a number.

If the argument is a *string-expression*, the result is the same number that would result from CAST(*string-expression* AS DOUBLE PRECISION). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant.

The single-byte character constant that must be used to delimit the decimal digits in *string-expression* from the whole part of the number is the default decimal point. For more information, see "Decimal point" on page 127.

The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: FLOAT is a synonym for DOUBLE_PRECISION and DOUBLE.

The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

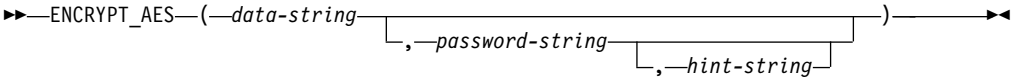
Example

- Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. To eliminate the possibility of out-of-range results, DOUBLE_PRECISION is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, DOUBLE_PRECISION(SALARY)/COMM
  FROM EMPLOYEE
  WHERE COMM > 0
```

ENCRYPT_AES

The ENCRYPT_AES function returns a value that is the result of encrypting *data-string* using the AES encryption algorithm. The password used for decryption is either the *password-string* value or the encryption password value (assigned by the SET ENCRYPTION PASSWORD statement).



data-string
An expression that returns the string value to be encrypted. The string expression must be a built-in string data type.

The length attribute for the data type of *data-string* is limited to 24 bytes (or 32 bytes) less than the maximum length of the result data type without a *hint-string* argument and 56 bytes (or 64 bytes) less than the maximum length of the result data type when the *hint-string* argument is specified.

password-string
An expression that returns a character string value with at least 6 bytes and no more than 127 bytes. The expression must not be a CLOB and the CCSID of the expression must not be 65535. The value represents the password used to encrypt the *data-string*. If the value of the password argument is null or not provided, the data will be encrypted using the ENCRYPTION PASSWORD value, which must have been set using the SET ENCRYPTION PASSWORD statement.

hint-string
An expression that returns a character string value with up to 32 bytes that will help data owners remember passwords (For example, 'Ocean' is a hint to remember 'Pacific'). The expression must not be a CLOB and the CCSID of the expression must not be 65535. If a hint value is specified, the hint is embedded into the result and can be retrieved using the GETHINT function. If the *password-string* is specified and this argument is the null value or not provided, no hint will be embedded in the result. If the *password-string* is not specified, the hint may be specified using the SET ENCRYPTION PASSWORD statement.

The data type of the result is determined by the first argument as shown in the following table:

Data Type of the First Argument	Data Type of the Result
BINARY or VARBINARY	VARBINARY
CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC	VARCHAR FOR BIT DATA
BLOB, CLOB, or DBCLOB	BLOB

The length attribute of the result depends on the arguments that are specified:

- When a *password-string* is specified but a *hint-string* is not specified, the length attribute of *data-string* plus 24 plus the number of bytes to a 16 byte boundary.
- Otherwise, the length attribute of *data-string* plus 64 plus the number of bytes to a 16 byte boundary.

The actual length of the result is the sum of :

- The actual length of *data-string* plus a number of bytes to get to a 16 byte boundary.
- The actual length of the hint.

  The actual length of the hint is zero if *hint-string* is not specified as a function argument or on the SET ENCRYPTION PASSWORD statement.
- *n*, where *n* (the amount of overhead necessary to encrypt the value) is 24 bytes (or 32 bytes if *data-string* is a LOB or different CCSID values are used for the *data-string*, the password, or the hint).

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Note that the encrypted result is longer than the *data-string* value. Therefore, when assigning encrypted values, ensure that the target is declared with sufficient size to contain the entire encrypted value.

Notes

Password protection: To prevent inadvertent access to the encryption password, do not specify *password-string* as a string constant in the source for a program, procedure, or function. Instead, use the SET ENCRYPTION PASSWORD statement or a host variable.

When connected to a remote relational database, the specified password itself is sent "in the clear". That is, the password itself is not encrypted. To protect the password in these cases, consider using a communications encryption mechanism such as IPSEC (or SSL if connecting between IBM i products).

Encryption algorithm: The internal encryption algorithm used is from the CLiC Toolkit from IBM Research. The 128-bit encryption key is derived from the password using a SHA1 message digest.

Encryption passwords and data: It is the user's responsibility to perform password management. Once the data is encrypted only the password used to encrypt it can be used to decrypt it. Be careful when using CHAR variables to set password values as they may be padded with blanks. The encrypted result may contain a null terminator and other non-printable characters.

Table column definition: When defining columns and distinct types to contain encrypted data:

- The column must be defined with a data type of CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, VARBINARY, or BLOB.
- The length attribute of the column must include an additional *n* bytes, where *n* is the overhead necessary to encrypt the data as described above.

Any assignment or cast to a column without one of these data types or with a length shorter than the suggested data length may result in an assignment error or, if the assignment is successful, a failure and lost data when the data is subsequently decrypted. Blanks are valid encrypted data values that may be truncated when stored in a column that is too short.

Some sample column length calculations:

```
Maximum length of non-encrypted data           6 bytes
Number of bytes to the next 16 byte boundary  10 bytes
Overhead                                      24 bytes (or 32 bytes)
```

```
                                                --------
Encrypted data column length                    40 bytes (or 48 bytes)


Maximum length of non-encrypted data            32 bytes
Number of bytes to a 16 byte boundary            0 bytes
Overhead                                        24 bytes (or 32 bytes)
                                                --------
Encrypted data column length                    56 bytes
```

Administration of encrypted data: Encrypted data can only be decrypted on servers that support the decryption functions that correspond to the ENCRYPT_AES function. Hence, replication of columns with encrypted data should only be done to servers that support the decryption functions.

Example

- Assume that table EMP1 has a social security column called SSN. This example uses the ENCRYPTION PASSWORD value to hold the encryption password.

```
SET ENCRYPTION PASSWORD = 'Ben123'

INSERT INTO EMP1 (SSN) VALUES ENCRYPT_AES( '289-46-8832' )
```

- This example explicitly passes the encryption password.

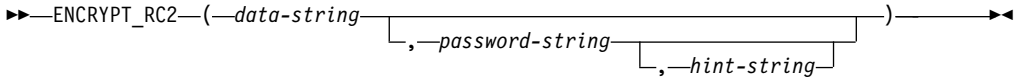
```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT_AES( '289-46-8832', 'Ben123' )
```

- The hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT_AES( '289-46-8832', 'Pacific', 'Ocean' )
```

ENCRYPT_RC2

The ENCRYPT_RC2 function returns a value that is the result of encrypting *data-string* using the RC2 encryption algorithm. The password used for decryption is either the *password-string* value or the encryption password value (assigned by the SET ENCRYPTION PASSWORD statement).



data-string
An expression that returns the string value to be encrypted. The string expression must be a built-in string data type.

The length attribute for the data type of *data-string* must be less than *m* - MOD(*m*,8) - *n* - 1, where *m* is the maximum length of the result data type and *n* is the amount of overhead necessary to encrypt the value.

- If a *hint-string* is not specified, *n* is 8 bytes.
- If a *hint-string* is specified, *n* is 40 bytes.

password-string
An expression that returns a character string value with at least 6 bytes and no more than 127 bytes. The expression must not be a CLOB. The value represents the password used to encrypt the *data-string*. If the value of the password argument is null or not provided, the data will be encrypted using the ENCRYPTION PASSWORD value, which must have been set using the SET ENCRYPTION PASSWORD statement.

hint-string
An expression that returns a character string value with up to 32 bytes that will help data owners remember passwords (For example, 'Ocean' is a hint to remember 'Pacific'). The expression must not be a CLOB. If a hint value is specified, the hint is embedded into the result and can be retrieved using the GETHINT function. If the *password-string* is specified and this argument is the null value or not provided, no hint will be embedded in the result. If the *password-string* is not specified, the hint may be specified using the SET ENCRYPTION PASSWORD statement.

The data type of the result is determined by the first argument as shown in the following table:

Data Type of the First Argument	Data Type of the Result
BINARY or VARBINARY	VARBINARY
CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC	VARCHAR FOR BIT DATA
BLOB, CLOB, or DBCLOB	BLOB

The length attribute of the result depends on the arguments that are specified:

- When a *password-string* is specified but a *hint-string* is not specified, the length attribute of *data-string* plus 16 plus the number of bytes to the next 8 byte boundary.[55]
- Otherwise, the length attribute of *data-string* plus 48 plus the number of bytes to the next 8 byte boundary.[55]

The actual length of the result is the sum of :

- The actual length of *data-string* plus a number of bytes to get to the next 8 byte boundary.[55]
- The actual length of the hint.

  The actual length of the hint is zero if *hint-string* is not specified as a function argument or on the SET ENCRYPTION PASSWORD statement.
- *n*, where *n* (the amount of overhead necessary to encrypt the value) is 8 bytes (or 16 bytes if *data-string* is a LOB or different CCSID values are used for the *data-string*, the password, or the hint).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note that the encrypted result is longer than the *data-string* value. Therefore, when assigning encrypted values, ensure that the target is declared with sufficient size to contain the entire encrypted value.

Notes

Password protection: To prevent inadvertent access to the encryption password, do not specify *password-string* as a string constant in the source for a program, procedure, or function. Instead, use the SET ENCRYPTION PASSWORD statement or a host variable.

When connected to a remote relational database, the specified password itself is sent "in the clear". That is, the password itself is not encrypted. To protect the password in these cases, consider using a communications encryption mechanism such as IPSEC (or SSL if connecting between IBM i products).

Encryption algorithm: The internal encryption algorithm used is RC2 block cipher with padding, the 128 bit secret key is derived from the password using a MD5 message digest.

Encryption passwords and data: It is the user's responsibility to perform password management. Once the data is encrypted only the password used to encrypt it can be used to decrypt it. Be careful when using CHAR variables to set password values as they may be padded with blanks. The encrypted result may contain a null terminator and other non-printable characters.

Table column definition: When defining columns and distinct types to contain encrypted data:

- The column must be defined with a data type of CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, VARBINARY, or BLOB.
- The length attribute of the column must include an additional *n* bytes, where *n* is the overhead necessary to encrypt the data as described above.

Any assignment or cast to a column without one of these data types or with a length shorter than the suggested data length may result in an assignment error or, if the assignment is successful, a failure and lost data when the data is subsequently decrypted. Blanks are valid encrypted data values that may be truncated when stored in a column that is too short.

55. Unlike ENCRYPT_TDES and ENCRYPT_AES, 8 bytes are added even if the length of *data-string* is already on an 8 byte boundary.

Some sample column length calculations:

```
Maximum length of non-encrypted data           6 bytes
Number of bytes to the next 8 byte boundary    2 bytes
Overhead                                       8 bytes (or 16 bytes)
                                              --------
Encrypted data column length                  16 bytes (or 32 bytes)


Maximum length of non-encrypted data          32 bytes
Number of bytes to the next 8 byte boundary    8 bytes
Overhead                                       8 bytes (or 16 bytes)
                                              --------
Encrypted data column length                  48 bytes (or 56 bytes)
```

Administration of encrypted data: Encrypted data can only be decrypted on servers that support the decryption functions that correspond to the ENCRYPT_RC2 function. Hence, replication of columns with encrypted data should only be done to servers that support the decryption functions.

Syntax alternatives: For compatibility with previous versions of DB2, ENCRYPT can be specified in place of ENCRYPT_RC2.

Example

- Assume that table EMP1 has a social security column called SSN. This example uses the ENCRYPTION PASSWORD value to hold the encryption password.

```
SET ENCRYPTION PASSWORD = 'Ben123'

INSERT INTO EMP1 (SSN) VALUES ENCRYPT_RC2( '289-46-8832' )
```

- This example explicitly passes the encryption password.

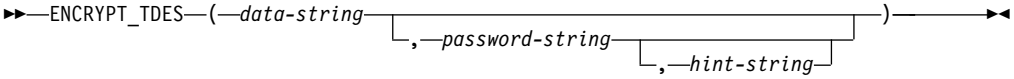
```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT_RC2( '289-46-8832', 'Ben123' )
```

- The hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT_RC2( '289-46-8832', 'Pacific', 'Ocean' )
```

ENCRYPT_TDES

The ENCRYPT_TDES function returns a value that is the result of encrypting *data-string* using the Triple DES encryption algorithm. The password used for decryption is either the *password-string* value or the encryption password value (assigned by the SET ENCRYPTION PASSWORD statement).



data-string
An expression that returns the string value to be encrypted. The string expression must be a built-in string data type.

The length attribute for the data type of *data-string* must be less than *m* - MOD(*m*,8) - *n* - 1, where *m* is the maximum length of the result data type and *n* is the amount of overhead necessary to encrypt the value.

password-string
An expression that returns a character string value with at least 6 bytes and no more than 127 bytes. The expression must not be a CLOB and the CCSID of the expression must not be 65535. The value represents the password used to encrypt the *data-string*. If the value of the password argument is null or not provided, the data will be encrypted using the ENCRYPTION PASSWORD value, which must have been set using the SET ENCRYPTION PASSWORD statement.

hint-string
An expression that returns a character string value with up to 32 bytes that will help data owners remember passwords (For example, 'Ocean' is a hint to remember 'Pacific'). The expression must not be a CLOB and the CCSID of the expression must not be 65535. If a hint value is specified, the hint is embedded into the result and can be retrieved using the GETHINT function. If the *password-string* is specified and this argument is the null value or not provided, no hint will be embedded in the result. If the *password-string* is not specified, the hint may be specified using the SET ENCRYPTION PASSWORD statement.

The data type of the result is determined by the first argument as shown in the following table:

Data Type of the First Argument	Data Type of the Result
BINARY or VARBINARY	VARBINARY
CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC	VARCHAR FOR BIT DATA
BLOB, CLOB, or DBCLOB	BLOB

The length attribute of the result depends on the arguments that are specified:

- When a *password-string* is specified but a *hint-string* is not specified, the length attribute of *data-string* plus 24 plus the number of bytes to an 8 byte boundary.
- Otherwise, the length attribute of *data-string* plus 56 plus the number of bytes to an 8 byte boundary.

The actual length of the result is the sum of :

- The actual length of *data-string* plus a number of bytes to get to the an 8 byte boundary.

- The actual length of the hint.

  The actual length of the hint is zero if *hint-string* is not specified as a function argument or on the SET ENCRYPTION PASSWORD statement.
- *n*, where *n* (the amount of overhead necessary to encrypt the value) is 16 bytes (or 24 bytes if *data-string* is a LOB or different CCSID values are used for the *data-string*, the password, or the hint).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note that the encrypted result is longer than the *data-string* value. Therefore, when assigning encrypted values, ensure that the target is declared with sufficient size to contain the entire encrypted value.

Notes

Password protection: To prevent inadvertent access to the encryption password, do not specify *password-string* as a string constant in the source for a program, procedure, or function. Instead, use the SET ENCRYPTION PASSWORD statement or a host variable.

When connected to a remote relational database, the specified password itself is sent "in the clear". That is, the password itself is not encrypted. To protect the password in these cases, consider using a communications encryption mechanism such as IPSEC (or SSL if connecting between IBM i products).

Encryption algorithm: The internal encryption algorithm used is Triple DES block cipher with padding, the 128 bit secret key is derived from the password using a SHA1 message digest.

Encryption passwords and data: It is the user's responsibility to perform password management. Once the data is encrypted only the password used to encrypt it can be used to decrypt it. Be careful when using CHAR variables to set password values as they may be padded with blanks. The encrypted result may contain a null terminator and other non-printable characters.

Table column definition: When defining columns and distinct types to contain encrypted data:

- The column must be defined with a data type of CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, VARBINARY, or BLOB.
- The length attribute of the column must include an additional *n* bytes, where *n* is the overhead necessary to encrypt the data as described above.

Any assignment or cast to a column without one of these data types or with a length shorter than the suggested data length may result in an assignment error or, if the assignment is successful, a failure and lost data when the data is subsequently decrypted. Blanks are valid encrypted data values that may be truncated when stored in a column that is too short.

Some sample column length calculations:

```
Maximum length of non-encrypted data           6 bytes
Number of bytes to the next 8 byte boundary    2 bytes
Overhead                                      16 bytes (or 24 bytes)
                                              --------
Encrypted data column length                  24 bytes (or 32 bytes)
```

```
Maximum length of non-encrypted data          32 bytes
Number of bytes to an 8 byte boundary          0 bytes
Overhead                                      16 bytes (or 24 bytes)
                                              --------
Encrypted data column length                  48 bytes (or 56 bytes)
```

Administration of encrypted data: Encrypted data can only be decrypted on servers that support the decryption functions that correspond to the ENCRYPT_TDES function. Hence, replication of columns with encrypted data should only be done to servers that support the decryption functions.

Example

- Assume that table EMP1 has a social security column called SSN. This example uses the ENCRYPTION PASSWORD value to hold the encryption password.

```
SET ENCRYPTION PASSWORD = 'Ben123'

INSERT INTO EMP1 (SSN) VALUES ENCRYPT_TDES( '289-46-8832' )
```

- This example explicitly passes the encryption password.

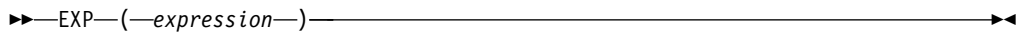
```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT_TDES( '289-46-8832', 'Ben123' )
```

- The hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT_TDES( '289-46-8832', 'Pacific', 'Ocean' )
```

EXP

The EXP function returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument. The EXP and LN functions are inverse operations.



expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

If the data type of the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: For decimal floating-point values the special values are treated as follows:

- EXP(NaN) returns NaN.
- EXP(-NaN) returns -NaN.
- EXP(Infinity) returns Infinity.
- EXP(-Infinity) returns 0.
- EXP(sNaN) and EXP(-sNaN) return a warning or error. [56]

Example

- Assume the host variable E is a DECIMAL(10,9) host variable with a value of 3.453789832.

```
SELECT EXP(:E)
  FROM SYSIBM.SYSDUMMY1
```

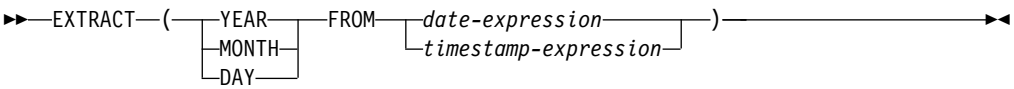
Returns the approximate value 31.62.

56. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN and -NaN are returned respectively with a warning.

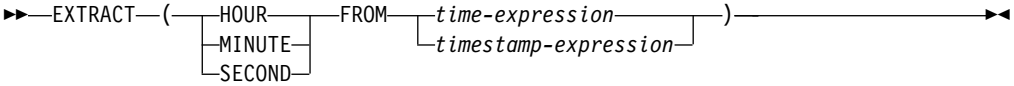
EXTRACT

The EXTRACT function returns a specified portion of a datetime value.

Extract Date Values



Extract Time Values



Extract Date Values

YEAR
Specifies that the year portion of the date or timestamp expression is returned. The result is identical to the YEAR scalar function. For more information, see "YEAR" on page 588.

MONTH
Specifies that the month portion of the date or timestamp expression is returned. The result is identical to the MONTH scalar function. For more information, see "MONTH" on page 416.

DAY
Specifies that the day portion of the date or timestamp expression is returned. The result is identical to the DAY scalar function. For more information, see "DAY" on page 309.

date-expression
An expression that returns the value of either a built-in date or built-in character or graphic string data type.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid character-string or graphic-string representation of a date. For the valid formats of string representations of dates, see "String representations of datetime values" on page 83.

timestamp-expression
An expression that returns the value of either a built-in timestamp or built-in character or graphic string data type.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid character-string or graphic-string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 83.

Extract Time Values

HOUR
Specifies that the hour portion of the time or timestamp expression is returned. The result is identical to the HOUR scalar function. For more information, see "HOUR" on page 376.

MINUTE
Specifies that the minute portion of the time or timestamp expression is returned. The result is identical to the MINUTE scalar function. For more information, see "MINUTE" on page 413.

SECOND
Specifies that the second portion of the time or timestamp expression is returned. If the expression is a *timestamp-expression*, the result is identical to the following:

```
DECIMAL((SECOND(expression) + DECIMAL(MICROSECOND(expression),12,6)/1000000), 8,6)
```

If the expression is a *time-expression*, the result is identical to the following:

```
DECIMAL((SECOND(expression), 8,6)
```

For more information, see "SECOND" on page 487 and "MICROSECOND" on page 410.

time-expression
An expression that returns the value of either a built-in time or built-in character or graphic string data type.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid character-string or graphic-string representation of a time. For the valid formats of string representations of times, see "String representations of datetime values" on page 83.

timestamp-expression
An expression that returns the value of either a built-in timestamp or built-in character or graphic string data type.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid character-string or graphic-string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 83.

The data type of the result of the function depends on the part of the datetime value that is specified:

- If YEAR, MONTH, DAY, HOUR, or MINUTE is specified, the data type of the result is INTEGER.
- If SECOND is specified, the data type of the result is DECIMAL(8,6). The fractional digits contains microseconds.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Assume the column PRSTDATE has an internal value equivalent to 1988-12-25.

```
SELECT EXTRACT( MONTH FROM PRSTDATE )
  FROM PROJECT
```

Results in the value 12.

FLOAT

The FLOAT function returns a floating point representation of a number or string.

Numeric to Float

String to Float

FLOAT is a synonym for the DOUBLE_PRECISION and DOUBLE functions. For more information, see "DOUBLE_PRECISION or DOUBLE" on page 344.

FLOOR

The FLOOR function returns the largest integer value less than or equal to *expression*.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is a decimal number. For example, an argument with a data type of DECIMAL(5,5) will result in DECIMAL(5,0).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: For decimal floating-point values, the special values are treated as follows:

- FLOOR(NaN) returns NaN.
- FLOOR(-NaN) returns -NaN.
- FLOOR(Infinity) returns Infinity.
- FLOOR(-Infinity) returns -Infinity.
- FLOOR(sNaN) and FLOOR(-sNaN) returns a warning or error.[57]

Example

- Use the FLOOR function to truncate any digits to the right of the decimal point.

```
SELECT FLOOR(SALARY)
  FROM EMPLOYEE
```

- Use FLOOR on both positive and negative numbers.

```
SELECT FLOOR( 3.5),
       FLOOR( 3.1),
       FLOOR(-3.1),
       FLOOR(-3.5)
  FROM SYSIBM.SYSDUMMY1
```

This example returns:

```
3.   3.  -4.  -4.
```

respectively.

57. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN and -NaN are returned respectively with a warning.

GENERATE_UNIQUE

The GENERATE_UNIQUE function returns a bit data character string 13 bytes long (CHAR(13) FOR BIT DATA) that is unique compared to any other execution of the same function. The function is defined as non-deterministic

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and the system serial number. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The sequence is based on the time when the function was executed.

This function differs from using the special register CURRENT TIMESTAMP in that a unique value is generated for each instance of the function in an SQL statement and each row of a multiple row insert statement, an insert statement with a fullselect, or an insert statement in a MERGE statement.

The timestamp value that is part of the result of this function can be determined using the TIMESTAMP function with the result of GENERATE_UNIQUE as an argument.

Examples

- Create a table that includes a column that is unique for each row. Populate this column using the GENERATE_UNIQUE function. Notice that the UNIQUE_ID column is defined as FOR BIT DATA to identify the column as a bit data character string.

```
CREATE TABLE EMP_UPDATE
  (UNIQUE_ID CHAR(13) FOR BIT DATA,
   EMPNO CHAR(6),
   TEXT VARCHAR(1000))
INSERT INTO EMP_UPDATE VALUES (GENERATE_UNIQUE(),
                               '000020',
                               'Update entry 1...')
INSERT INTO EMP_UPDATE VALUES (GENERATE_UNIQUE(),
                               '000050',
                               'Update entry 2...')
```

  This table will have a unique identifier for each row provided that the UNIQUE_ID column is always set using GENERATE_UNIQUE. This can be done by introducing a trigger on the table.

```
CREATE TRIGGER  EMP_UPDATE_UNIQUE
  NO CASCADE BEFORE INSERT ON EMP_UPDATE
  REFERENCING NEW AS NEW_UPD
  FOR EACH ROW MODE DB2SQL
  SET NEW_UPD.UNIQUE_ID = GENERATE_UNIQUE()
```

  With this trigger, the previous INSERT statements that were used to populate the table can be issued without specifying a value for the UNIQUE_ID column:

```
INSERT INTO (EMPNO, TEXT) EMP_UPDATE VALUES ('000020', 'Update entry 1...')
INSERT INTO (EMPNO, TEXT) EMP_UPDATE VALUES ('000050', 'Update entry 2...')
```

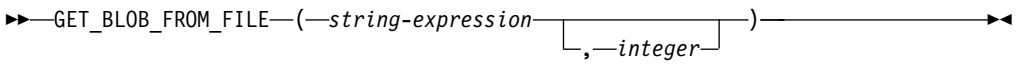
  The timestamp (in UTC) for when a row was added to EMP_UPDATE can be returned using:

```
SELECT TIMESTAMP(UNIQUE_ID), EMPNO, TEXT FROM EMP_UPDATE
```

Therefore, the table does not need a timestamp column to record when a row is inserted.

GET_BLOB_FROM_FILE

The GET_BLOB_FROM_FILE function returns the data from a source stream file or a source physical file.



string-expression
The argument must be a string expression that specifies a path and file name. The file name may be a name of a source stream file or a source physical file. For a source physical file, the string must be in the form 'library/file(member)'.

integer
An integer constant that specifies how to handle trailing whitespace when the specified file is a source physical file. Valid values are:

0	whitespace is returned as it exists in the file
1	trailing whitespace (blanks) is removed except for the first trailing blank

If this argument is not specified for a source physical file, the default is 0. If it is specified for a source stream file, it is ignored.

The result of the function is a BLOB locator.

The function will read the file specified by the argument with no CCSID conversion and return it as a BLOB locator. The function must be run under commitment control. The locator will be freed when a COMMIT or ROLLBACK is performed.

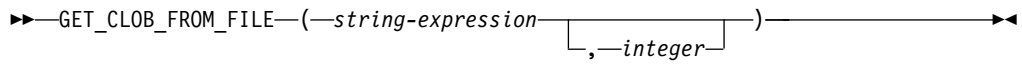
Examples

Register an XML schema document in the XSR registry where the XML schema is in a source stream file.

```
CALL XSR_REGISTER ('myschemalib', 'myschema', NULL,
                   GET_BLOB_FROM_FILE('/home/XML/MySchema.XSD',0), NULL)
```

GET_CLOB_FROM_FILE

The GET_CLOB_FROM_FILE function returns the data from a source stream file or a source physical file.



string-expression
The argument must be a string expression that specifies a path and file name. The file name may be a name of a source stream file or a source physical file. For a source physical file, the string must be in the form 'library/file(member)'.

integer
An integer constant that specifies how to handle trailing whitespace when the specified file is a source physical file. Valid values are:

0	whitespace is returned as it exists in the file
1	trailing whitespace (blanks) is removed except for the first trailing blank

If this argument is not specified for a source physical file, the default is 0. If it is specified for a source stream file, it is ignored.

The result of the function is a CLOB locator.

The function will read the file specified by the argument, convert the data to the default job CCSID, and return it as a CLOB locator. The function must be run under commitment control. The locator will be freed when a COMMIT or ROLLBACK is performed.

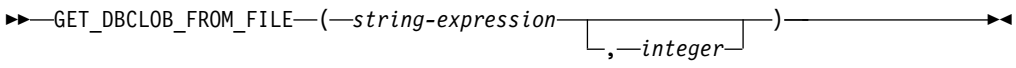
Examples

Assign the data contained in the source file to host variable HV1. The data will be converted to the default job CCSID and trailing blanks from each line of the source file will be removed.

```
SET :HV1 = GET_CLOB_FROM_FILE('MYLIB/MYFILE(MYMBR)',1)
```

GET_DBCLOB_FROM_FILE

The GET_DBCLOB_FROM_FILE function returns the data from a source stream file or a source physical file.



string-expression
The argument must be a string expression that specifies a path and file name. The file name may be a name of a source stream file or a source physical file. For a source physical file, the string must be in the form 'library/file(member)'.

integer
An integer constant that specifies how to handle trailing whitespace when the specified file is a source physical file. Valid values are:

0	whitespace is returned as it exists in the file
1	trailing whitespace (blanks) is removed except for the first trailing blank

If this argument is not specified for a source physical file, the default is 0. If it is specified for a source stream file, it is ignored.

The result of the function is a DBCLOB locator.

The function will read the file specified by the argument, convert the data to the double-byte CCSID associated with the default CCSID, and return it as a DBCLOB locator. The function must be run under commitment control. The locator will be freed when a COMMIT or ROLLBACK is performed.

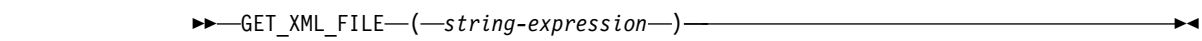
Examples

Assign the data contained in the source stream file to host variable HV1. The data will be converted to the double-byte CCSID associated with the default CCSID.

```
SET :HV1 = GET_DBCLOB_FROM_FILE('/home/XML/MySchema.XSD')
```

GET_XML_FILE

The GET_XML_FILE function returns the data from a source stream file or a source physical file.



string-expression
The argument must be a string expression that specifies a path and file name. The file name may be a name of a source stream file or a source physical file. For a source physical file, the string must be in the form 'library/file(member)'.

The result of the function is a BLOB locator.

The function will read the file specified by the argument and convert the data to UTF-8. If the file does not contains an XML declaration, one will be added. It will return it as a BLOB locator. The function must be run under commitment control. The locator will be freed when a COMMIT or ROLLBACK is performed.

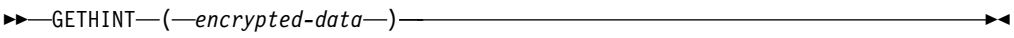
Examples

Register an XML schema document in the XSR registry where the XML schema is in a source stream file.

```
CALL XSR_REGISTER ('myschemalib', 'myschema', NULL,
                   GET_XML_FILE('/home/XML/MySchema.XSD'), NULL)
```

GETHINT

The GETHINT function will return the password hint if one is found in the *encrypted-data*. A password hint is a phrase that will help data owners remember passwords (For example, 'Ocean' as a hint to remember 'Pacific').



encrypted-data
An expression that must be a string expression that returns a complete, encrypted data value of a CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, VARBINARY, or BLOB built-in data type. The data string must have been encrypted using the ENCRYPT_RC2 or ENCRYPT_TDES function.

The data type of the result is VARCHAR(32). The actual length of the result is the actual length of the hint that was provided when the data was encrypted.

The result can be null. If the argument is null or if a hint was not added to the *encrypted-data* by the ENCRYPT_RC2 or ENCRYPT_TDES function, the result is the null value.

The CCSID of the result is the default CCSID of the current server.

Example

- The hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT_RC2( '289-46-8832', 'Pacific', 'Ocean' )

SELECT GETHINT( SSN )
  FROM EMP1
```

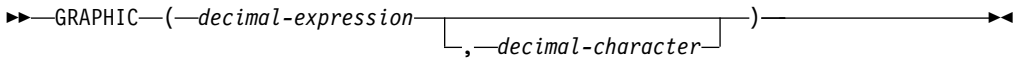
  The GETHINT function returns the original hint value 'Ocean'.

GRAPHIC

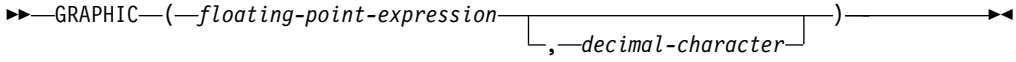
The GRAPHIC function returns a fixed-length graphic-string representation of a string expression.

Integer to Graphic

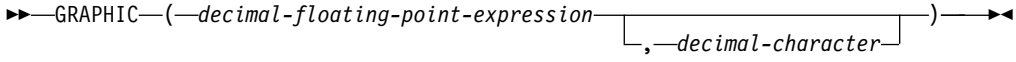
Decimal to GRAPHIC



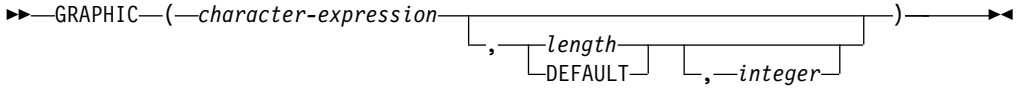
Floating-point to GRAPHIC



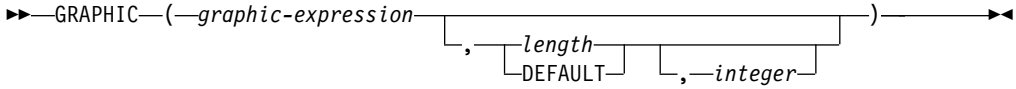
Decimal floating-point to GRAPHIC



Character to Graphic



Graphic to Graphic



The GRAPHIC function returns a graphic-string representation of:

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number if the first argument is a packed or zoned decimal number
- A double-precision floating-point number if the first argument is a DOUBLE or REAL
- A decimal floating-point number if the first argument is a DECFLOAT
- A character string if the first argument is any type of character string
- A graphic string if the first argument is any type of graphic string

The result of the function is a fixed-length graphic string (GRAPHIC).

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Integer to Graphic

integer-expression
An expression that returns a value that is an integer data type (either SMALLINT, INTEGER, or BIGINT).

The result is a fixed-length graphic string of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. It is left justified.

- If the argument is a small integer, the length attribute of the result is 6.
- If the argument is a large integer, the length attribute of the result is 11.
- If the argument is a big integer, the length attribute of the result is 20.

The result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit or the *decimal-character*.

The CCSID of the result is 1200 (UTF-16).

Decimal to Graphic

decimal-expression
An expression that returns a value that is a packed or zoned decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is wanted, the DECIMAL scalar function can be used to make the change.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a fixed-length graphic string representation of the argument. The result includes a decimal character and up to *p* digits, where *p* is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length attribute of the result is 2+*p* where *p* is the precision of the *decimal-expression*. The result is the smallest number of characters that can be used to represent the result. Leading zeros are not included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit or the *decimal-character*.

The CCSID of the result is 1200 (UTF-16).

Floating-point to Graphic

floating-point expression
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a fixed-length graphic string representation of the argument in the form of a floating-point constant.

The length attribute of the result is 24. The result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0E0.

The CCSID of the result is 1200 (UTF-16).

Decimal floating-point to Graphic

decimal-floating-point expression
An expression that returns a value that is a built-in decimal floating-point data type.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a fixed-length graphic string representation of the argument in the form of a floating-point constant.

The length attribute of the result is 42. The actual length of the result is the smallest number of characters that represents the value of the argument, including the sign, digits, and *decimal-character*. Trailing zeros are significant. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0.

If the DECFLOAT value is Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, a minus sign will be the first character in the string. The DECFLOAT special value sNaN does not result in an exception when converted to a string.

The CCSID of the result is 1200 (UTF-16).

Character to Graphic

character-expression
An expression that returns a value that is a built-in character-string data type. It cannot be a CHAR or VARCHAR bit data. If the expression is an empty string or the EBCDIC string `X'0E0F'`, the result is a single double-byte blank.

length
An integer constant that specifies the length attribute of the result and must be an integer constant between 1 and 16383 if the first argument is not nullable or between 1 and 16382 if the first argument is nullable. If the length of *character-expression* is less than the length specified, the result is padded with double-byte blanks to the length of the result.

If *length* is not specified, or if DEFAULT is specified, the length attribute of the result is the same as the length attribute of the first argument.

Each character of the argument determines a character of the result. If the length attribute of the resulting fixed-length string is less than the actual length of the first argument, truncation is performed and no warning is returned.

integer
An integer constant that specifies the CCSID of the result. It must be a DBCS, UTF-16, or UCS-2 CCSID. The CCSID cannot be 65535. If the CCSID represents Unicode graphic data, each character of the argument determines a character of the result. The nth character of the result is the UTF-16 or UCS-2 equivalent of the nth character of the argument.

If *integer* is not specified then the CCSID of the result is determined by a mixed CCSID. Let M denote that mixed CCSID.

In the following rules, S denotes one of the following:

- If the string expression is a host variable containing data in a foreign encoding scheme, S is the result of the expression after converting the data to a CCSID in a native encoding scheme. (See "Character conversion" on page 32 for more information.)
- If the string expression is data in a native encoding scheme, S is that string expression.

M is determined as follows:

- If the CCSID of S is a mixed CCSID, M is that CCSID.
- If the CCSID of S is an SBCS CCSID:
  - If the CCSID of S has an associated mixed CCSID, M is that CCSID.
  - Otherwise the operation is not allowed.

The following table summarizes the result CCSID based on M.

M	Result CCSID	Description	DBCS Substitution Character
930	300	Japanese EBCDIC	X'FEFE'
933	834	Korean EBCDIC	X'FEFE'
935	837	S-Chinese EBCDIC	X'FEFE'
937	835	T-Chinese EBCDIC	X'FEFE'
939	300	Japanese EBCDIC	X'FEFE'
5026	4396	Japanese EBCDIC	X'FEFE'
5035	4396	Japanese EBCDIC	X'FEFE'

The equivalence of SBCS and DBCS characters depends on M. Regardless of the CCSID, every double-byte code point in the argument is considered a DBCS character, and every single-byte code point in the argument is considered an SBCS character with the exception of the EBCDIC mixed data shift codes `X'0E'` and `X'0F'`.

- If the nth character of the argument is a DBCS character, the nth character of the result is that DBCS character.
- If the nth character of the argument is an SBCS character that has an equivalent DBCS character, the nth character of the result is that equivalent DBCS character.
- If the nth character of the argument is an SBCS character that does not have an equivalent DBCS character, the nth character of the result is the DBCS substitution character.

Graphic to Graphic

graphic-expression
An expression that returns a value of a built-in graphic-string data type.

length
An integer constant that specifies the length attribute of the result and must be an integer constant between 1 and 16383 if the first argument is not nullable or between 1 and 16382 if the first argument is nullable. If the length of *graphic-expression* is less than the length specified, the result is padded with double-byte blanks to the length of the result.

If the second argument is not specified, or if DEFAULT is specified, the length attribute of the result is the same as the length attribute of the first argument.

If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID of the result. It must be a DBCS, UTF-16, or UCS-2 CCSID. The CCSID cannot be 65535.

If *integer* is not specified then the CCSID of the result is the CCSID of the first argument.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the first argument is a string and the length attribute is specified. For more information, see "CAST specification" on page 185.

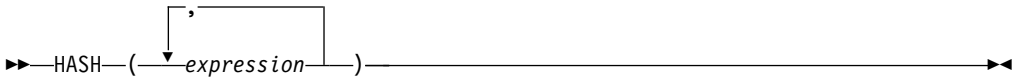
Example

- Using the EMPLOYEE table, set the host variable DESC (GRAPHIC(24)) to the GRAPHIC equivalent of the first name (FIRSTNME) for employee number (EMPNO) '000050'.

```
SELECT GRAPHIC( VARGRAPHIC(FIRSTNME))
  INTO :DESC
  FROM EMPLOYEE
  WHERE EMPNO = '000050'
```

HASH

The HASH function returns the partition number of a set of values.



Also see "HASHED_VALUE" on page 373. For more information about partition numbers, see the DB2 Multisystem topic collection.

expression
An expression that returns a value of any built-in data type except date, time, timestamp, floating-point, XML, or DataLink values.

The result of the function is a large integer with a value between 0 and 1023.

If any of the arguments are null, the result is zero. The result cannot be null.

Example

- Use the HASH function to determine what the partitions would be if the partitioning key was composed of EMPNO and LASTNAME. This query returns the partition number for every row in EMPLOYEE.

```
SELECT HASH(EMPNO, LASTNAME)
  FROM EMPLOYEE
```

HASHED_VALUE

The HASHED_VALUE function returns the partition map index number of a row obtained by applying the hashing function on the partitioning key value of the row.

Also see the "HASH" on page 372 function. If the argument identifies a non-distributed table, the value 0 is returned. For more information about partition maps and partitioning keys, see the DB2 Multisystem topic collection.

table-designator
A table designator of the subselect. For more information about table designators, see "Table designators" on page 143.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

If the argument identifies a view, common table expression, or nested table expression, the function returns the partition map index number of its base table. If the argument identifies a view, common table expression, or nested table expression derived from more than one base table, the function returns the partition map index number of the first table in the outer subselect of the view, common table expression, or nested table expression.

The argument must not identify a view, common table expression, or nested table expression whose outer fullselect subselect includes an aggregate function, a GROUP BY clause, a HAVING clause, a UNION clause, an INTERSECT clause, DISTINCT clause, or VALUES clause. The HASHED_VALUE function cannot be specified in a SELECT clause if the fullselect contains an aggregate function, a GROUP BY clause, a HAVING clause, or a VALUES clause. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is a large integer with a value between 0 and 1023. The result can be null.

Note

Syntax alternatives: PARTITION is a synonym for HASHED_VALUE.

Example

- Select the employee number (EMPNO) from the EMPLOYEE table for all rows where the partition map index number is equal to 100.

```
SELECT EMPNO
  FROM EMPLOYEE
  WHERE HASHED_VALUE(EMPLOYEE) = 100
```

HEX

The HEX function returns a hexadecimal representation of a value.

expression
An expression that returns a value of any built-in data type other than a character or binary string with a length attribute greater than 16 336 or a graphic string with a length attribute greater than 8168.

The result of the function is a character string. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is a string of hexadecimal digits, the first two digits represent the first byte of the argument, the next two digits represent the second byte of the argument, and so forth. If the argument is a datetime value, the result is the hexadecimal representation of the internal form of the argument.[58]

If the argument is not a graphic string, the actual length of the result is twice the length of the argument. If the argument is a graphic string, the actual length of the result is four times the length of the argument. If the data type of the result is varying length, the length is limited to the maximum length of the data type. The length of the argument is the value that would be returned if the argument were passed to the LENGTH scalar function. For more information, see "LENGTH" on page 392.

The data type and length attribute of the result depends on the attributes of the argument:

- If the argument is not a string, the result is CHAR with a length attribute that is twice the length of the argument.
- If the argument is a fixed-length character string with a length attribute that is less than one half the maximum length attribute of CHAR, the result is CHAR with a length attribute that is twice the length attribute of the argument. If the argument is a fixed-length graphic string with a length attribute that is less than one fourth the maximum length attribute of CHAR, the result is CHAR with a length attribute that is four times the length attribute of the argument. For more information about the product-specific maximum length, see Table 113 on page 1497.
- Otherwise, the result is VARCHAR whose length attribute depends on the following:
  - If the argument is a character or binary string, the length attribute of the result is the minimum of twice the length attribute of the argument and the maximum length of the data type.
  - If the argument is a graphic string, the length attribute of the result is the minimum of four times the length attribute of the argument and the maximum length of the data type.

The length attribute of the result cannot be greater than the product-specific length attribute of CHAR or VARCHAR. See Table 113 on page 1497 for more information.

58. This hexadecimal representation for DATE, TIMESTAMP, and NUMERIC data types is different from other database products because the internal form for these data types is different.

The CCSID of the string is the default SBCS CCSID at the current server.

Example

- Use the HEX function to return a hexadecimal representation of the education level for each employee.

```
SELECT FIRSTNME, MIDINIT, LASTNAME, HEX(EDLEVEL)
  FROM EMPLOYEE
```

HOUR

The HOUR function returns the hour part of a value.

expression
An expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a time or timestamp. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 83.
- If *expression* is a number, it must be a time duration or timestamp duration. For the valid formats of datetime durations, see "Datetime operands and durations" on page 173.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a time, timestamp, or valid character-string representation of a time or timestamp:

  The result is the hour part of the value, which is an integer between 0 and 24.
- If the argument is a time duration or timestamp duration:

  The result is the hour part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example

- Using the CL_SCHED sample table, select all the classes that start in the afternoon.

```
SELECT *
  FROM CL_SCHED
  WHERE HOUR(STARTING) BETWEEN 12 AND 17
```

IDENTITY_VAL_LOCAL

IDENTITY_VAL_LOCAL is a non-deterministic function that returns the most recently assigned value for an identity column.

The function has no input parameters. The result is a DECIMAL(31,0) regardless of the actual data type of the identity column that the result value corresponds to.

The value returned is the value that was assigned to the identity column of the table identified in the most recent insert operation (specified in either an INSERT statement or a MERGE statement). The insert operation has to be issued at the same level; that is, the value has to be available locally within the level at which it was assigned until replaced by the next assigned value. A new level is initiated when a trigger, function, or stored procedure is invoked. A trigger condition is at the same level as the associated triggered action.

The assigned value can be a value supplied by the user (if the identity column is defined as GENERATED BY DEFAULT) or an identity value that was generated by the database manager.

The result can be null. The result is null if an insert operation has not been issued for a table containing an identity column at the current processing level. This includes invoking the function in a before or after insert trigger.

The result of the IDENTITY_VAL_LOCAL function is not affected by the following statements:

- An insert operation for a table which does not contain an identity column
- An UPDATE statement
- A COMMIT statement
- A ROLLBACK statement

Notes

The following notes explain the behavior of the function when it is invoked in various situations:

Invoking the function within the VALUES clause of an insert operation
Expressions in an insert operation are evaluated before values are assigned to the target columns of the insert operation. Thus, when you invoke IDENTITY_VAL_LOCAL in an insert operation, the value that is used is the most recently assigned value for an identity column from a previous insert operation. The function returns the null value if no such insert operation had been executed within the same level as the invocation of the IDENTITY_VAL_LOCAL function.

Invoking the function following a failed insert operation
The function returns an unpredictable result when it is invoked after the unsuccessful execution of an insert operation for a table with an identity column. The value might be the value that would have been returned from the function had it been invoked before the failed insert operation or the value that would have been assigned had the insert operation succeeded. The actual value returned depends on the point of failure and is therefore unpredictable.

Invoking the function within the SELECT statement of a cursor
Because the results of the IDENTITY_VAL_LOCAL function are not deterministic, the result of an invocation of the IDENTITY_VAL_LOCAL function from within the SELECT statement of a cursor can vary for each FETCH statement.

Invoking the function within the trigger condition of an insert trigger
The result of invoking the IDENTITY_VAL_LOCAL function from within the condition of an insert trigger is the null value.

Invoking the function within a triggered action of an insert trigger
Multiple before or after insert triggers can exist for a table. In such cases, each trigger is processed separately, and identity values generated by SQL statements issued within a triggered action are not available to other triggered actions using the IDENTITY_VAL_LOCAL function. This is the case even though the multiple triggered actions are conceptually defined at the same level.

Do not use the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger. The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of a before insert trigger is the null value. The value for the identity column of the table for which the trigger is defined cannot be obtained by invoking the IDENTITY_VAL_LOCAL function within the triggered action of a before insert trigger. However, the value for the identity column can be obtained in the triggered action by referencing the trigger transition variable for the identity column.

The result of invoking the IDENTITY_VAL_LOCAL function in the triggered action of an after insert trigger is the value assigned to an identity column of the table identified in the most recent insert operation invoked in the same triggered action for a table containing an identity column. If an insert operation for a table containing an identity column was not executed within the same triggered action before invoking the IDENTITY_VAL_LOCAL function, then the function returns a null value.

Invoking the function following an insert operation with triggered actions
The result of invoking the function after an insert operation that activates triggers is the value actually assigned to the identity column (that is, the value that would be returned on a subsequent SELECT statement). This value is not necessarily the value provided in the insert operation or a value generated by the database manager. The assigned value could be a value that was specified in a SET transition variable statement within the triggered action of a before insert trigger for a trigger transition variable associated with the identity column.

Scope of IDENTITY_VAL_LOCAL
The IDENTITY_VAL_LOCAL value persists until the next insert operation in the current session into a table that has an identity column defined on it, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements. The IDENTITY_VAL_LOCAL value cannot be directly set and is a result of inserting a row into a table.

A technique commonly used, especially for performance, is for an application or product to manage a set of connections and route transactions to an arbitrary connection. In these situations, the availability of the IDENTITY_VAL_LOCAL value should only be relied on until the end of the transaction.

Alternative to IDENTITY_VAL_LOCAL:
It is recommended that a SELECT FROM INSERT be used to obtain the assigned value for an identity column. See "table-reference" on page 633 for more information.

Examples

- Set the variable IVAR to the value assigned to the identity column in the EMPLOYEE table. The value returned from the function in the VALUES INTO statement should be 1.

```
CREATE TABLE EMPLOYEE
  (EMPNO INTEGER GENERATED ALWAYS AS IDENTITY,
   NAME CHAR(30),
   SALARY DECIMAL(5,2),
   DEPTNO SMALLINT)

INSERT INTO EMPLOYEE
  (NAME, SALARY, DEPTNO)
  VALUES('Rupert', 989.99, 50)

VALUES IDENTITY_VAL_LOCAL() INTO  :IVAR
```

- Assume two tables, T1 and T2, have an identity column named C1. The database manager generates values 1, 2, 3,...for the C1 column in table T1, and values 10, 11, 12,...for the C1 column in table T2.

```
CREATE TABLE T1
  (C1 SMALLINT GENERATED ALWAYS AS IDENTITY,
   C2 SMALLINT)

CREATE TABLE T2
  (C1 DECIMAL(15,0) GENERATED BY DEFAULT AS IDENTITY ( START WITH 10 ) ,
   C2 SMALLINT)

INSERT INTO T1 ( C2 ) VALUES(5)

INSERT INTO T1 ( C2 ) VALUES(5)

SELECT  * FROM T1
```

C1	C2
1	5
2	5

```
VALUES IDENTITY_VAL_LOCAL() INTO  :IVAR
```

At this point, the IDENTITY_VAL_LOCAL function would return a value of 2 in IVAR. The following INSERT statement inserts a single row into T2 where column C2 gets a value of 2 from the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T2 ( C2 ) VALUES( IDENTITY_VAL_LOCAL() )

SELECT  * FROM T2
  WHERE C1 = DECIMAL( IDENTITY_VAL_LOCAL(), 15, 0)
```

C1	C2
10	2

Invoking the IDENTITY_VAL_LOCAL function after this INSERT would result in a value of 10, which is the value generated by the database manager for column C1 of T2. Assume another single row is inserted into T2. For the following INSERT statement, the database manager assigns a value of 13 to

identity column C1 and gives C2 a value of 10 from IDENTITY_VAL_LOCAL. Thus, C2 is given the last identity value that was inserted into T2.

```
INSERT INTO T2 ( C2, C1 ) VALUES( IDENTITY_VAL_LOCAL(), 13 )

SELECT  * FROM T2
  WHERE C1 = DECIMAL( IDENTITY_VAL_LOCAL(), 15, 0)
```

C1	C2
13	10

- The IDENTITY_VAL_LOCAL function can also be invoked in an INSERT statement that both invokes the IDENTITY_VAL_LOCAL function and causes a new value for an identity column to be assigned. The next value to be returned is thus established when the IDENTITY_VAL_LOCAL function is invoked after the INSERT statement completes. For example, consider the following table definition:

  ```
  CREATE TABLE T3
    (C1 SMALLINT GENERATED BY DEFAULT AS IDENTITY,
     C2 SMALLINT)
  ```

  For the following INSERT statement, specify a value of 25 for the C2 column, and the database manager generates a value of 1 for C1, the identity column. This establishes 1 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

  ```
  INSERT INTO T3 ( C2 ) VALUES( 25 )
  ```

  In the following INSERT statement, the IDENTITY_VAL_LOCAL function is invoked to provide a value for the C2 column. A value of 1 (the identity value assigned to the C1 column of the first row) is assigned to the C2 column, and the database manager generates a value of 2 for C1, the identity column. This establishes 2 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

  ```
  INSERT INTO T3 ( C2 ) VALUES( IDENTITY_VAL_LOCAL() )
  ```

  In the following INSERT statement, the IDENTITY_VAL_LOCAL function is again invoked to provide a value for the C2 column, and the user provides a value of 11 for C1, the identity column. A value of 2 (the identity value assigned to the C1 column of the second row) is assigned to the C2 column. The assignment of 11 to C1 establishes 11 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

  ```
  INSERT INTO T3 ( C2, C1 ) VALUES( IDENTITY_VAL_LOCAL(), 11 )
  ```

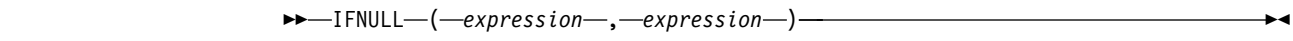
  After the 3 INSERT statements have been processed, table T3 contains the following:

C1	C2
1	25
2	1
11	2

  The contents of T3 illustrate that the expressions in the VALUES clause are evaluated before the assignments for the columns of the INSERT statement. Thus, an invocation of an IDENTITY_VAL_LOCAL function invoked from a VALUES clause of an INSERT statement uses the most recently assigned value for an identity column in a previous INSERT statement.

IFNULL

The IFNULL function returns the value of the first non-null expression.



The IFNULL function is identical to the COALESCE scalar function with two arguments. For more information, see "COALESCE" on page 292.

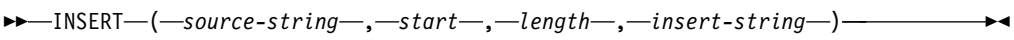
Example

- When selecting the employee number (EMPNO) and salary (SALARY) from all the rows in the EMPLOYEE table, if the salary is missing (that is, null), then return a value of zero.

```
SELECT EMPNO, IFNULL(SALARY,0)
  FROM EMPLOYEE
```

INSERT

Returns a string where *length* characters have been deleted from *source-string* beginning at *start* and where *insert-string* has been inserted into *source-string* beginning at *start*.



source-string
An expression that specifies the source string. The *source-string* may be any built-in numeric or string expression. It must be compatible with the *insert-string*. For more information about data type compatibility, see "Assignments and comparisons" on page 98. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531. The actual length of the string must be greater than zero.

start
An expression that returns a built-in BIGINT, INTEGER, or SMALLINT data type. The integer specifies the starting character within *source-string* where the deletion of characters and the insertion of another string is to begin. The value of the integer must be in the range of 1 to the length of *source-string* plus one.

length
An expression that returns a built-in BIGINT, INTEGER, or SMALLINT data type. The integer specifies the number of characters that are to be deleted from *source-string*, starting at the character position identified by *start*. The value of the integer must be in the range of 0 to the length of *source-string*.

insert-string
An expression that specifies the string to be inserted into *source-string*, starting at the position identified by *start*. The *insert-string* may be any built-in numeric or string expression. It must be compatible with the *source-string*. For more information about data type compatibility, see "Assignments and comparisons" on page 98. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531. The actual length of the string must be greater than zero.

The data type of the result of the function depends on the data type of the first and fourth arguments. The result data type is the same as if the two arguments were concatenated except that the result is always a varying-length string. For more information see "Conversion rules for operations that combine strings" on page 120.

The length attribute of the result depends on the arguments:

- If *start* and *length* are constants, the length attribute of the result is:

```
L1 - MIN((L1-V2 + 1), V3) + L4
```

  where:

```
L1 is the length attribute of source-string
V2 depends on the encoding schema of source-string:
- If the source-string is UTF-8, the value MIN(L1+1,start*3)
- If the source-string is mixed data, the value MIN(L1+1,(start-1)*2.5+4)
- Otherwise, the value of start
V3 is the value of length
L4 is the length attribute of insert-string
```

- Otherwise, the length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string*.

If the length attribute of the result exceeds the maximum for the result data type, an error is returned.

The actual length of the result is:

```
A1 - MIN((A1 -V2 + 1), V3) + A4
```

where:

```
A1 is the actual length of source-string
V2 is the value of start
V3 is the value of length
A4 is the actual length of insert-string
```

If the actual length of the result string exceeds the maximum for the result data type, an error is returned.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is determined by the CCSID of *source-string* and *insert-string*. The resulting CCSID is the same as if the two arguments were concatenated. For more information, see "Conversion rules for operations that combine strings" on page 120.

Examples

- The following example shows how the string 'INSERTING' can be changed into other strings. The use of the CHAR function limits the length of the resulting string to 10 characters.

```
SELECT INSERT('INSERTING', 4, 2, 'IS'),
       INSERT('INSERTING', 4, 0, 'IS'),
       INSERT('INSERTING', 4, 2, '')
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 'INSISTING ', 'INSISERTIN', and 'INSTING '.

- The previous example demonstrated how to insert text into the middle of some text. This example shows how to insert text before some text by using 1 as the starting point (*start*).

```
SELECT INSERT('INSERTING', 1, 0, 'XX'),
       INSERT('INSERTING', 1, 1, 'XX'),
       INSERT('INSERTING', 1, 2, 'XX'),
       INSERT('INSERTING', 1, 3, 'XX')
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 'XXINSERTIN', 'XXNSERTING', 'XXSERTING ', and 'XXERTING '.

- The following example shows how to insert text after some text. Add 'XX' at the end of string 'ABCABC'. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

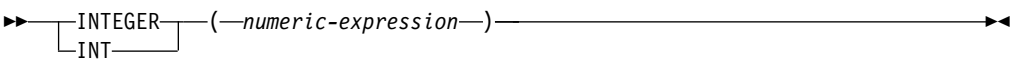
```
SELECT INSERT('ABCABC', 7, 0, 'XX')
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 'ABCABCXX '.

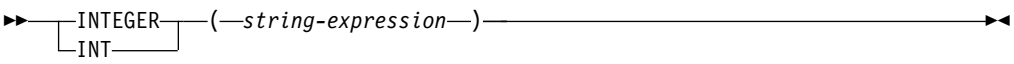
INTEGER or INT

The INTEGER function returns an integer representation.

Numeric to Integer



String to Integer



The INTEGER function returns an integer representation of:

- A number
- A character or graphic string representation of a decimal number
- A character or graphic string representation of an integer
- A character or graphic string representation of a floating-point number
- A character or graphic string representation of a decimal floating-point number

Numeric to Integer

numeric-expression
An expression that returns a numeric value of any built-in numeric data type.

If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of integers, an error is returned. The fractional part of the argument is truncated.

String to Integer

string-expression
An expression that returns a value that is a character-string or graphic-string representation of a number.

If the argument is a *string-expression*, the result is the same number that would result from CAST(*string-expression* AS INTEGER). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant. If the whole part of the argument is not within the range of integers, an error is returned. Any fractional part of the argument is truncated.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

Example

- Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and the employee number (EMPNO).

```
SELECT INTEGER(SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
  FROM EMPLOYEE
```

JULIAN_DAY

The JULIAN_DAY function returns an integer value representing a number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date specified in the argument.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Julian and Gregorian calendar: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is not taken into account by this function.

Examples

- Using sample table EMPLOYEE, set the integer host variable JDAY to the Julian day of the day that Christine Haas (EMPNO = '000010') was employed (HIREDATE = '1965-01-01').

```
SELECT JULIAN_DAY(HIREDATE)
  INTO :JDAY
  FROM EMPLOYEE
  WHERE EMPNO = '000010'
```

  The result is that JDAY is set to 2438762.

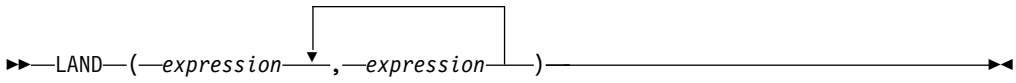
- Set integer host variable JDAY to the Julian day for January 1, 1998.

```
SELECT JULIAN_DAY('1998-01-01')
  INTO :JDAY
  FROM SYSIBM.SYSDUMMY1
```

  The result is that JDAY is set to 2450815.

LAND

The LAND function returns a string that is the logical 'AND' of the argument strings. This function takes the first argument string, does an AND operation with the next string, and then continues to do AND operations with each successive argument using the previous result. If a character-string argument is shorter than the previous result, it is padded with blanks. If a binary-string argument is shorter than the previous result, it is padded with hexadecimal zeros.



The arguments must be compatible.

expression
An expression that returns a value of any built-in numeric or string data type, but cannot be LOBs. The arguments cannot be mixed data character strings, UTF-8 character strings, or graphic strings. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The arguments are converted, if necessary, to the attributes of the result. The attributes of the result are determined as follows:

- If all the arguments are fixed-length strings, the result is a fixed-length string of length *n*, where *n* is the length of the longest argument.
- If any argument is a varying-length string, the result is a varying-length string with length attribute *n*, where *n* is the length attribute of the argument with greatest length attribute. The actual length of the result is *m*, where *m* is the actual length of the longest argument.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The CCSID of the result is 65535.

Example

- Assume the host variable L1 is a CHARACTER(2) host variable with a value of X'A1B1', host variable L2 is a CHARACTER(3) host variable with a value of X'F0F040', and host variable L3 is a CHARACTER(4) host variable with a value of X'A1B10040'.

```
SELECT LAND(:L1,:L2,:L3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value X'A0B00040'.

LAST_DAY

The LAST_DAY function returns a date or timestamp that represents the last day of the month indicated by *expression*.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a timestamp if *expression* is a timestamp. Otherwise, the result of the function is a date. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Set the host variable END_OF_MONTH with the last day of the current month.

  ```
  SET :END_OF_MONTH = LAST_DAY(CURRENT_DATE)
  ```

  The host variable END_OF_MONTH is set with the value representing the end of the current month. If the current day is 2000-02-10, then END_OF_MONTH is set to 2000-02-29.
- Set the host variable END_OF_MONTH with the last day of the month in EUR format for the given date.

  ```
  SET :END_OF_MONTH = CHAR(LAST_DAY('1965-07-07'), EUR)
  ```

  The host variable END_OF_MONTH is set with the value '31.07.1965'.
- Assuming that the default date format is ISO,

  ```
  SELECT LAST_DAY('2000-04-24')
  FROM SYSIBM.SYSDUMMY1
  ```

  Returns '2000–04–30' which is the last day of April in 2000.

LCASE

The LCASE function returns a string in which all the characters have been converted to lowercase characters, based on the CCSID of the argument.

The LCASE function is identical to the LOWER function. For more information, see "LOWER" on page 403.

LEFT

The LEFT function returns the leftmost *integer* characters of *expression*.

If *expression* is a character string, the result is a character string. If *expression* is a graphic string, the result is a graphic string. If *expression* is a binary string, the result is a binary string.

expression
An expression that specifies the string from which the result is derived. The arguments must be expressions that return a value of any built-in numeric, character string, graphic string, or a binary string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

A substring of *expression* is zero or more contiguous characters of *expression*. If *expression* is a character string or graphic string, a single character is either an SBCS, DBCS, or multiple-byte character. If *expression* is a binary string, the result is the number of bytes in the argument.

integer
An expression that returns a built-in integer data type. The integer specifies the length of the result. The value of *integer* must be greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *expression*.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *expression* and a data type that depends on the data type of *expression*:

Data type of *expression*	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The actual length of the result is *integer*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of *expression*.

Example

- Assume the host variable NAME (VARCHAR(50)) has a value of 'KATIE AUSTIN' and the host variable FIRSTNAME_LEN (int) has a value of 5.

```
SELECT LEFT(:NAME, :FIRSTNAME_LEN)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'KATIE'

- Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'.

```
SELECT LEFT(NAME, 2), SUBSTR(NAME, 1, 2)
  FROM T1
  WHERE NAME = 'Jürgen'
```

  Returns the value 'Jü' for LEFT and 'JÊ' for SUBSTR(NAME, 1, 2).

LENGTH

The LENGTH function returns the length of a value.

See "CHARACTER_LENGTH" on page 285, "OCTET_LENGTH" on page 437, and "BIT_LENGTH" on page 274 for similar functions.

expression
An expression that returns a value of any built-in data type.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is the length of the argument. The length of strings includes blanks. The length of a varying-length string is the actual length, not the length attribute.

The length of a graphic string is the number of double-byte characters (the number of bytes divided by 2). The length of all other values is the number of bytes used to represent the value:

- 2 for small integer
- 4 for large integer
- 8 for big integer
- The integral part of (p/2)+1 for packed decimal numbers with precision p
- p for zoned decimal numbers with precision p
- 4 for single-precision float
- 8 for double-precision float
- 8 for DECFLOAT(16)
- 16 for DECFLOAT(34)
- The length of the string for strings
- 3 for time
- 4 for date
- 10 for timestamp
- The actual number of bytes used to store the DataLink value (plus 19 if the DataLink is FILE LINK CONTROL and READ PERMISSION DB) for datalinks
- 26 for row ID

Examples

- Assume the host variable ADDRESS is a varying-length character string with a value of '895 Don Mills Road'.

```
SELECT LENGTH(:ADDRESS)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 18.

- Assume that PRSTDATE is a column of type DATE.

```
SELECT LENGTH(PRSTDATE)
  FROM PROJECT
```

  Returns the value 4.

- Assume that PRSTDATE is a column of type DATE.

```
SELECT LENGTH(CHAR(PRSTDATE, EUR))
  FROM PROJECT
```

Returns the value 10.

LN

The LN function returns the natural logarithm of a number. The LN and EXP functions are inverse operations.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. The value of the argument must be greater than zero.

If the data type of the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: For decimal floating-point values the special values are treated as follows:

- LN(NaN) returns NaN. [59]
- LN(-NaN) returns NaN. [59]
- LN(Infinity) returns Infinity.
- LN(-Infinity) returns NaN. [59]
- LN(sNaN) and LN(-sNaN) return a warning or error. [59]
- LN(0) returns -Infinity.
- LN with a negative argument, including -Infinity, returns NaN. [59]

Example

- Assume the host variable NATLOG is a DECIMAL(4,2) host variable with a value of 31.62.

```
SELECT LN(:NATLOG)
  FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 3.45.

59. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN is returned with a warning.

LNOT

The LNOT function returns a string that is the logical NOT of the argument string.

expression
An expression that returns a value of any built-in numeric or string data type, but cannot be LOBs. The arguments cannot be mixed data character strings, UTF-8 character strings, or graphic strings. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The data type and length attribute of the result is the same as the data type and length attribute of the argument value. If the argument is a varying-length string, the actual length of the result is the same as the actual length of the argument value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the result is 65535.

Example

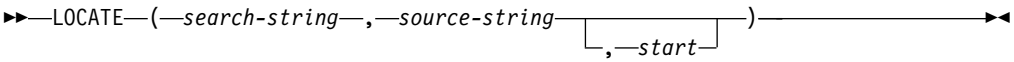
- Assume the host variable L1 is a CHARACTER(2) host variable with a value of X'F0F0'.

```
SELECT LNOT(:L1)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value X'0F0F'.

LOCATE

The LOCATE function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*. If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin.



search-string
An expression that specifies the string that is to be searched for. *search-string* may be any built-in numeric, datetime, or string expression. It must be compatible with the *source-string*. A numeric or datetime argument is cast to a character string before evaluating the function. For more information about converting numeric and datetime to a character string, see "VARCHAR" on page 531.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* may be any built-in numeric, datetime, or string expression. A numeric or datetime argument is cast to a character string before evaluating the function. For more information about converting numeric and datetime to a character string, see "VARCHAR" on page 531.

start
An expression that specifies the position within *source-string* at which the search is to start. *start* may be any built-in numeric, character string, or graphic string expression. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be greater than zero.

If *start* is specified, the function is similar to:

```
POSITION( SUBSTRING(source-string,start) , search-string ) + start - 1
```

If *start* is not specified, the function is equivalent to:

```
POSITION( source-string , search-string )
```

For more information, see "POSITION" on page 442.

The result of the function is a large integer. If any of the arguments can be null, the result can be null; if any of the arguments is null, the result is the null value.

The LOCATE function operates on a character basis. Because LOCATE operates on a character-string basis, any shift-in and shift-out characters are not required to be in exactly the same position and their only significance is to indicate which characters are SBCS and which characters are DBCS.

If the CCSID of the *search-string* is different than the CCSID of the *source-string*, it is converted to the CCSID of the *source-string*.

If a collating sequence other than *HEX is in effect when the statement that contains the LOCATE function is executed and the arguments are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted

values for each value in the set. The weighted values are based on the collating sequence. An ICU collating sequence table may not be specified with the LOCATE function.

Example

- Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD' within the NOTE_TEXT column for all entries in the IN_TRAY table that contain these words.

```
SELECT RECEIVED, SUBJECT, LOCATE('GOOD', NOTE_TEXT)
  FROM IN_TRAY
  WHERE LOCATE('GOOD', NOTE_TEXT) <> 0
```

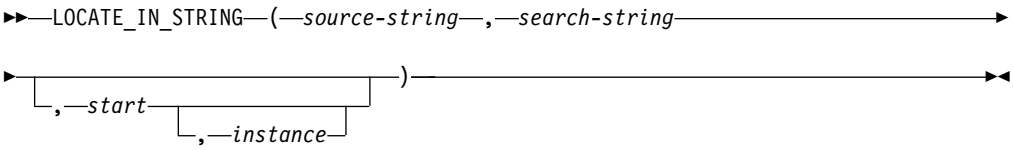
- Assume that NOTE is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen lives on Hegelstraße'. Find the character position of the character 'ß' in the string.

```
SELECT LOCATE( 'ß', NOTE ), POSSTR( NOTE_TEXT, 'ß')
  FROM T1
```

  Returns the value 26 for LOCATE and 27 for POSSTR.

LOCATE_IN_STRING

The LOCATE_IN_STRING function returns the starting position of a string (called the *search-string*) within another string (called the *source-string*). If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*. If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin.



If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin. If *start* is specified, an optional *instance* number can also be specified. The *instance* argument is used to determine the specific occurrence of *search-string* within *source-string*. Each unique instance can include any of the characters in a previous instance, but not all characters in a previous instance.

If the *search-string* has a length of zero, the result returned by the function is 1. If the *source-string* has a length of zero, the result returned by the function is 0. If neither condition exists, and if the value of *search-string* is equal to an identical length of a substring of contiguous positions within the value of *source-string*, the result returned by the function is the starting position of that substring within the *source-string* value; otherwise, the result returned by the function is 0.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* may be any built-in numeric, datetime, or string expression. A numeric or datetime argument is cast to a character string before evaluating the function. For more information about converting numeric and datetime to a character string, see "VARCHAR" on page 531.

search-string
An expression that specifies the string that is to be searched for. *search-string* may be any built-in numeric, datetime, or string expression. It must be compatible with the *source-string*. A numeric or datetime argument is cast to a character string before evaluating the function. For more information about converting numeric and datetime to a character string, see "VARCHAR" on page 531.

start
An expression that specifies the position within *source-string* at which the search is to start. *start* may be any built-in numeric, character string, or graphic string expression. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If the value of the integer is greater than zero, the search begins at *start* and continues for each position to the end of the string. If the value of the integer is less than zero, the search begins at CHARACTER_LENGTH(*source-string*) + *start* + 1 and continues for each position to the beginning of the string.

If *start* is not specified, the function is equivalent to:

```
POSITION( source-string , search-string )
```

If *start* is zero, an error is returned.

instance
An expression that specifies which instance of *search-string* to search for within *source-string*. The expression must return a value that is a built-in numeric, character string, or graphic string data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. If *instance* is not specified, the default is 1. The value of the integer must be greater than or equal to 1.

At each search position, a match is found when the substring at that position and CHARACTER_LENGTH(*search-string*) - 1 values to the right of the search position in *source-string*, is equal to *search-string*.

The result of the function is a large integer. The result is the starting position of the instance of *search-string* within *source-string*. The value is relative to the beginning of the string (regardless of the specification of *start*).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The LOCATE_IN_STRING function operates on a character basis. Because LOCATE_IN_STRING operates on a character-string basis, any shift-in and shift-out characters are not required to be in exactly the same position and their only significance is to indicate which characters are SBCS and which characters are DBCS.

If the CCSID of the *search-string* is different than the CCSID of the *source-string*, it is converted to the CCSID of the *source-string*.

If a collating sequence other than *HEX is in effect when the statement that contains the LOCATE_IN_STRING function is executed and the arguments are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the collating sequence. An ICU collating sequence table may not be specified with the LOCATE function.

INSTR can be used as a synonym for LOCATE_IN_STRING.

Example

- Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße' by searching from the end of the string, and set the host variable POSITION with the position within the string.

```
SET :POSITION = LOCATE_IN_STRING('Jürgen lives on Hegelstraße','ß',-1);
```

  The value of host variable POSITION is set to 26.

- Find the position of an occurrence of the character 'N' in the string 'WINNING' by searching from the start of the string.

```
SELECT LOCATE_IN_STRING('WINNING','N',1,3),
       LOCATE_IN_STRING('WINNING','N',3,2),
       LOCATE_IN_STRING('WINNING','N',3,3)
FROM SYSIBM.SYSDUMMY1;
```

  Returns the values:

```
6     4     6
```

- Find the position of an occurrence of the character 'N' in the string 'WINNING' by searching from the end of the string.

```
SELECT LOCATE_IN_STRING('WINNING','N',-1,3),
       LOCATE_IN_STRING('WINNING','N',-3,2),
       LOCATE_IN_STRING('WINNING','N',-3,3)
FROM SYSIBM.SYSDUMMY1;
```

  Returns the values:

```
3     3     0
```

LOG10

The LOG10 function returns the common logarithm (base 10) of a number. The LOG10 and ANTILOG functions are inverse operations.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

If the data type of the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: For decimal floating-point values the special values are treated as follows:

- LOG10(NaN) returns NaN. [60]
- LOG10(-NaN) returns NaN. [60]
- LOG10(Infinity) returns Infinity.
- LOG10(-Infinity) returns NaN. [60]
- LOG10(sNaN) and LOG10(-sNaN) return a warning or error. [60]
- LOG10(0) returns -Infinity.
- LOG10 with a negative argument, including -Infinity, returns NaN. [60]

Syntax alternatives: LOG is a synonym for LOG10. It is supported only for compatibility with previous DB2 releases. LOG10 should be used instead of LOG because some database managers and applications implement LOG as the natural logarithm of a number instead of the common logarithm of a number.

Example

- Assume the host variable L is a DECIMAL(4,2) host variable with a value of 31.62.

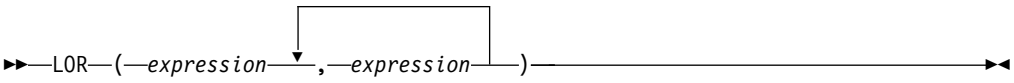
```
SELECT LOG10(:L)
  FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 1.49.

60. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN is returned with a warning.

LOR

The LOR function returns a string that is the logical OR of the argument strings. This function takes the first argument string, does an OR operation with the next string, and then continues to do OR operations for each successive argument using the previous result. If a character-string argument is shorter than the previous result, it is padded with blanks. If a binary-string argument is shorter than the previous result, it is padded with hexadecimal zeros.



The arguments must be compatible.

expression
An expression that returns a value of any built-in numeric or string data type, but cannot be LOBs. The arguments cannot be mixed data character strings, UTF-8 character strings, or graphic strings. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The arguments are converted, if necessary, to the attributes of the result. The attributes of the result are determined as follows:

- If all the arguments are fixed-length strings, the result is a fixed-length string of length *n*, where *n* is the length of the longest argument.
- If any argument is a varying-length string, the result is a varying-length string with length attribute *n*, where *n* is the length attribute of the argument with greatest length attribute. The actual length of the result is *m*, where *m* is the actual length of the longest argument.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The CCSID of the result is 65535.

Example

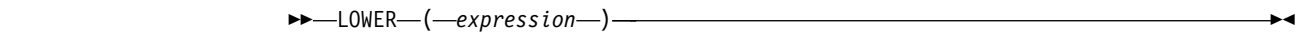
- Assume the host variable L1 is a CHARACTER(2) host variable with a value of X'0101', host variable L2 is a CHARACTER(3) host variable with a value of X'F0F000', and host variable L3 is a CHARACTER(4) host variable with a value of X'0000000F'.

```
SELECT LOR(:L1,:L2,:L3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value X'F1F1404F'.

LOWER

The LOWER function returns a string in which all the characters have been converted to lowercase characters, based on the CCSID of the argument. Only SBCS, Unicode graphic characters are converted. The characters A-Z are converted to a-z, and characters with diacritical marks are converted to their lowercase equivalent, if any.



Refer to the UCS-2 level 1 mapping tables topic in theGlobalization topic collection for a description of the monocasing tables that are used for this translation.

expression
An expression that specifies the string to be converted. *expression* must be any built-in numeric, character, Unicode graphic string. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The result of the function has the same data type, length attribute, actual length, and CCSID as the argument. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Note

Syntax alternatives: LCASE is a synonym for LOWER.

Examples

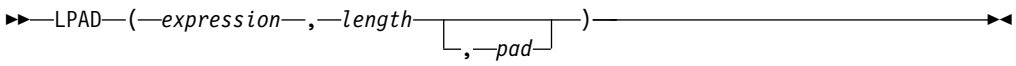
- Ensure that the characters in the value of host variable NAME are lowercase. NAME has a data type of VARCHAR(30) and a value of 'Christine Smith'.

```
SELECT LOWER(:NAME)
    FROM SYSIBM.SYSDUMMY1
```

  The result is the value 'christine smith'.

LPAD

The LPAD function returns a string composed of *expression* that is padded on the left.



The LPAD function treats leading or trailing blanks in *expression* as significant. Padding will only occur if the actual length of *expression* is less than *length*, and *pad* is not an empty string.

expression
An expression that specifies the string from which the result is derived.

Expression must be a built-in string, numeric, or datetime data type. A numeric or datetime argument is cast to VARCHAR with a CCSID that is the default SBCS CCSID at the current server before evaluating the function. For more information about converting numeric or datetime to a varying character string, see "VARCHAR" on page 531.

length
An expression that specifies the length of the result. The expression must return a value that is a built-in numeric, character-string, or graphic-string data type. If the data type of the expression is not INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be zero or a positive integer that is less than or equal to *n*, where *n* is the maximum length of the result data type. See Appendix A, "SQL limits," on page 1493 for more information.

If *expression* is a graphic string, *length* indicates the number of DBCS or Unicode graphic characters. If *expression* is a character string, *length* indicates the number of characters where a character may consist of one or more bytes. If *expression* is a binary string, *length* indicates the number of bytes.

pad
An expression that specifies the string with which to pad. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is a numeric or datetime data type, it is implicitly cast to VARCHAR with a CCSID that is the default SBCS CCSID at the current server before evaluating the function.

If *pad* is not specified, the pad character is set as follows:

- For character and graphic strings, a single-byte, double-byte, UTF-16, or UTF-8 blank character based on the data type and CCSID of *expression*.[61]
- For binary strings, hexadecimal zeros.

The value for *expression* and the value for *pad* must have compatible data types. If the CCSID of *pad* is different than the CCSID of *expression*, the *pad* value is converted to the CCSID of *expression*. For more information about data type compatibility, see "Assignments and comparisons" on page 98.

The data type of the result depends on the data type of *expression*:

61. UTF-16 or UCS-2 defines a blank character at code point X'0020' and X'3000'. The database manager pads with the blank at code point X'0020'. The database manager pads UTF-8 with a blank at code point X'20'

Data type of *expression*	Data Type of the Result for LPAD
CHAR or VARCHAR or numeric or datetime	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The length attribute of the result depends on *length*. If *length* is explicitly specified by an integer constant that is greater than zero, the length attribute of the result is *length*. If *length* is explicitly specified by an integer constant that is zero, the length attribute of the result is 1. If *length* is specified as an expression, the length attribute of the result is the minimum of *m*+100 and the maximum length of the result data type, where *m* is the length attribute of *expression*. See Appendix A, “SQL limits,” on page 1493 for more information.

The actual length of the result is determined from *length*.

- If *length* is 0, the actual length is 0 and the result is the empty result string.
- If *length* is equal to the actual length of *expression*, the actual length is the length of *expression*.
- If *length* is less than the actual length of *expression*, the result is truncated. The actual length is *length* unless the result data type is varying-length mixed data or varying-length Unicode. In this case, only complete characters will be truncated.
  - For Unicode data, the actual length may be *length*-1 to prevent a double-byte character from being split.
  - For mixed data the actual length may be as little as *length*-3 to account for truncation of a double byte character and possibly a “shift-in” character (X'0F') and a “shift-out” character (X'0E').
- If *length* is greater than the actual length of *expression*, the actual length is *length* unless the result data type is varying-length mixed data or varying-length Unicode and *pad* contains double-byte characters. In this case, only complete characters will be padded.
  - For Unicode data, the actual length may be *length*-1 to prevent a double-byte character from being split.
  - For mixed data, the actual length may be may be as little as *length*-3 to account for truncation of a double byte character and possibly a “shift-in” character (X'0F') and a “shift-out” character (X'0E'). Also, this result will not have redundant shift codes “at the seam”. Thus, if the pad is a string ending with a “shift-in” character (X'0F'), and expression begins with a “shift-out” character (X'0E'), these two bytes are eliminated from the result.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of *expression*.

Examples

- *Example 1*: Assume that NAME is a VARCHAR(15) column that contains the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the left with periods:

```
SELECT LPAD(NAME,15,'.' ) AS NAME FROM T1;
```

returns:

```
NAME
---------------
.........Chris
...........Meg
..........Jeff
```

- *Example 2*: Assume that NAME is a VARCHAR(15) column that contains the values "Chris", "Meg", and "Jeff". The following query will only pad each value to a length of 5:

```
SELECT LPAD(NAME,5,'.' ) AS NAME FROM T1;
```

returns:

```
NAME
-----
Chris
..Meg
.Jeff
```

- *Example 3*: Assume that NAME is a CHAR(15) column containing the values "Chris", "Meg", and "Jeff". The LPAD function does not pad because NAME is a fixed length character field and is blank padded already. However, since the length of the result is 5, the columns are truncated:

```
SELECT LPAD(NAME,5,'.' ) AS NAME FROM T1;
```

returns:

```
NAME
-----
Chris
Meg
Jeff
```

- *Example 4*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". In some cases, a partial instance of the pad specification is returned:

```
SELECT LPAD(NAME,15,'123' ) AS NAME FROM T1;
```

returns:

```
NAME
---------------
1231231231Chris
123123123123Meg
12312312312Jeff
```

- *Example 5*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that "Chris" is truncated, "Meg" is padded, and "Jeff" is unchanged:

```
SELECT LPAD(NAME,4,'.' ) AS NAME FROM T1;
```

returns:

```
NAME
----
Chri
.Meg
Jeff
```

LTRIM

The LTRIM function removes blanks or hexadecimal zeros from the beginning of an expression.

expression
An expression that returns a value of any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function.[62] For more information about converting numeric to a character string, see "VARCHAR" on page 531.

- If the argument is a binary string, the leading hexadecimal zeros (X'00') are removed.
- If the argument is a DBCS graphic string, the leading DBCS blanks are removed.
- If the first argument is a Unicode graphic string, the leading UTF-16 or UCS-2 blanks are removed.
- If the first argument is a UTF-8 character string, the leading UTF-8 blanks are removed.
- Otherwise, leading SBCS blanks are removed.

The data type of the result depends on the data type of *expression*:

Data type of *expression*	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The length attribute of the result is the same as the length attribute of *expression*. The actual length of the result is the length of *expression* minus the number of bytes removed. If all characters are removed, the result is an empty string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The CCSID of the result is the same as that of the string.

Example

- Assume the host variable HELLO of type CHAR(9) has a value of ' Hello'.

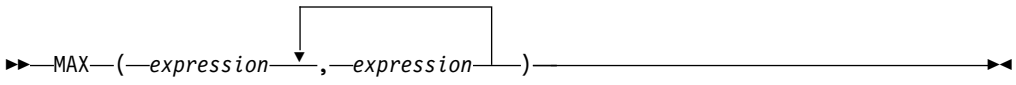
```
SELECT LTRIM(:HELLO)
  FROM SYSIBM.SYSDUMMY1
```

Results in: 'Hello'.

62. The LTRIM function returns the same results as: STRIP(*expression*,LEADING)

MAX

The MAX scalar function returns the maximum value in a set of values.



The arguments must be compatible. Character-string arguments are compatible with datetime values. The arguments cannot be DataLink or XML values.

expression
An expression that returns the value of any built-in numeric or string data type. If one of the arguments is numeric, then character and graphic string arguments are cast to numeric before evaluating the function.

The result of the function is the largest argument value. The result can be null if at least one argument can be null; the result is the null value if one of the arguments is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands as explained in "Rules for result data types" on page 115.

If a collating sequence other than *HEX is in effect when the statement is executed and the arguments are SBCS data, mixed data, or Unicode data, the weighted values of the strings are compared instead of the actual values. The weighted values are based on the collating sequence.

Examples

- Assume the host variable M1 is a DECIMAL(2,1) host variable with a value of 5.5, host variable M2 is a DECIMAL(3,1) host variable with a value of 4.5, and host variable M3 is a DECIMAL(3,2) host variable with a value of 6.25.

```
SELECT MAX(:M1,:M2,:M3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 6.25.

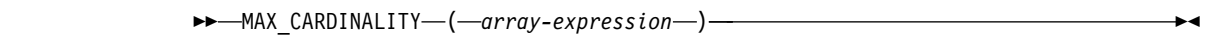
- Assume the host variable M1 is a CHARACTER(2) host variable with a value of 'AA', host variable M2 is a CHARACTER(3) host variable with a value of 'AA ', and host variable M3 is a CHARACTER(4) host variable with a value of 'AA A'.

```
SELECT MAX(:M1,:M2,:M3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'AA A'.

MAX_CARDINALITY

The MAX_CARDINALITY function returns a value representing the maximum number of elements an array can contain. This is the cardinality specified on the CREATE TYPE (Array) statement for the user-defined array type.



array-expression
The expression can be either an SQL procedure variable or parameter of an array data type, or a cast specification of a parameter marker to an array data type.

The result of the function is BIGINT. The result cannot be null.

Example

Assume that type PHONE_LIST is defined as:

```
CREATE TYPE PHONE_LIST AS INTEGER ARRAY[100]
```

The array NUMBERS is of type PHONE_LIST. The following SET statement assigns variable CARD the value 100, in accordance with the definition of PHONE_LIST:

```
SET CARD = MAX_CARDINALITY(NUMBERS);
```

MICROSECOND

The MICROSECOND function returns the microsecond part of a value.

expression
An expression that returns a value of one of the following built-in data types: a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 83.
- If *expression* is a number, it must be a timestamp duration. For the valid formats of datetime durations, see "Datetime operands and durations" on page 173.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a timestamp or a valid character-string representation of a timestamp:

  The result is the microsecond part of the value, which is an integer between 0 and 999999.
- If the argument is a duration:

  The result is the microsecond part of the value, which is an integer between -999999 and 999999. A nonzero result has the same sign as the argument.

Example

- Assume a table TABLEA contains two columns, TS1 and TS2, of type TIMESTAMP. Select all rows in which the microseconds portion of TS1 is not zero and the seconds portion of TS1 and TS2 are identical.

```
SELECT *
  FROM TABLEA
  WHERE MICROSECOND(TS1) <> 0 AND SECOND(TS1) = SECOND(TS2)
```

MIDNIGHT_SECONDS

The MIDNIGHT_SECONDS function returns an integer value that is greater than or equal to 0 and less than or equal to 86 400 representing the number of seconds between midnight and the time value specified in the argument.

expression
An expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, or a graphic string. It must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 83.

The result of the function is large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Find the number of seconds between midnight and 00:01:00, and midnight and 13:10:10. Assume that host variable XTIME1 has a value of '00:01:00', and that XTIME2 has a value of '13:10:10'.

```
SELECT MIDNIGHT_SECONDS(:XTIME1), MIDNIGHT_SECONDS(:XTIME2)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 60 and 47410. Because there are 60 seconds in a minute and 3600 seconds in an hour, 00:01:00 is 60 seconds after midnight ((60 * 1) + 0), and 13:10:10 is 47410 seconds ((3600 * 13) + (60 * 10) + 10).

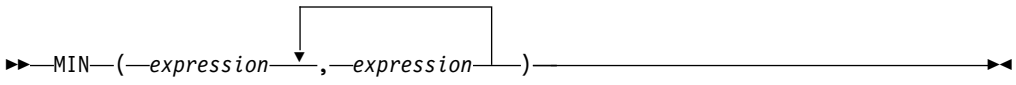
- Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

```
SELECT MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00')
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 86400 and 0. Although these two values represent the same point in time, different values are returned.

MIN

The MIN scalar function returns the minimum value in a set of values.



The arguments must be compatible. Character-string arguments are compatible with datetime values. The arguments cannot be DataLink or XML values.

expression
An expression that returns a value of any built-in numeric or string data type. If one of the arguments is numeric, then character and graphic string arguments are cast to numeric before evaluating the function.

The result of the function is the minimum argument value. The result can be null if at least one argument can be null; the result is the null value if one of the arguments is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands as explained in "Rules for result data types" on page 115.

If a collating sequence other than *HEX is in effect when the statement is executed and the arguments are SBCS data, mixed data, or Unicode data, the weighted values of the strings are compared instead of the actual values. The weighted values are based on the collating sequence.

Examples

- Assume the host variable M1 is a DECIMAL(2,1) host variable with a value of 5.5, host variable M2 is a DECIMAL(3,1) host variable with a value of 4.5, and host variable M3 is a DECIMAL(3,2) host variable with a value of 6.25.

```
SELECT MIN(:M1,:M2,:M3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 4.50.

- Assume the host variable M1 is a CHARACTER(2) host variable with a value of 'AA', host variable M2 is a CHARACTER(3) host variable with a value of 'AAA', and host variable M3 is a CHARACTER(4) host variable with a value of 'AAAA'.

```
SELECT MIN(:M1,:M2,:M3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'AA '.

MINUTE

The MINUTE function returns the minute part of a value.

expression
An expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 83.
- If *expression* is a number, it must be a time duration or timestamp duration. For the valid formats of datetime durations, see "Datetime operands and durations" on page 173.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a time, a timestamp, or a valid character-string representation of a time or timestamp:

  The result is the minute part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration:

  The result is the minute part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

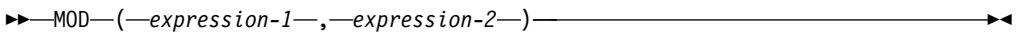
Example

- Using the CL_SCHED sample table, select all classes with a duration less than 50 minutes.

```
SELECT *
  FROM CL_SCHED
  WHERE HOUR(ENDING - STARTING) = 0 AND
      MINUTE(ENDING - STARTING) < 50
```

MOD

The MOD function divides the first argument by the second argument and returns the remainder.



The formula used to calculate the remainder is:

```
MOD(x,y) = x - (x/y) * y
```

where x/y is the truncated integer result of the division. The result is negative only if first argument is negative.

expression-1
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

expression-2
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. *expression-2* cannot be zero unless either argument is decimal floating-point.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The attributes of the result are determined as follows:

- If both arguments are large or small integers with zero scale, the data type of the result is large integer.
- If both arguments are integers with zero scale and at least one of the arguments is a big integer, the data type of the result is big integer.
- If one argument is an integer with zero scale and the other is decimal, the result is decimal with the same precision and scale as the decimal argument.
- If both arguments are decimal or integer with scale numbers, the result is decimal. The precision of the result is min (p-s,p'-s') + max (s,s'), and the scale of the result is max (s,s'), where the symbols p and s denote the precision and scale of the first operand, and the symbols p' and s' denote the precision and scale of the second operand.
- If either argument is decimal floating-point, the data type of the result is DECFLOAT(34). If the argument is a special decimal floating-point value, the general rules for arithmetic operations apply. See "General arithmetic operation rules for DECFLOAT" on page 169 for more information.
- If either argument is floating point, the data type of the result is double-precision floating point.

  The operation is performed in floating point; the operands having been first converted to double-precision floating-point numbers, if necessary.

  An operation involving a floating-point number and an integer is performed with a temporary copy of the integer that has been converted to double-precision floating point. An operation involving a floating-point number

and a decimal number is performed with a temporary copy of the decimal number that has been converted to double-precision floating point. The result of a floating-point operation must be within the range of floating-point numbers.

If either argument is decimal floating-point and the second operand evaluates to 0, the result is NaN and the invalid operation warning (SQLSTATE 0168D) is issued.[63] MOD(1, -Infinity) returns the value 1.

Examples

- Assume the host variable M1 is an integer host variable with a value of 5, and host variable M2 is an integer host variable with a value of 2.

```
SELECT MOD(:M1,:M2)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 1.

- Assume the host variable M1 is an integer host variable with a value of 5, and host variable M2 is a DECIMAL(3,2) host variable with a value of 2.20.

```
SELECT MOD(:M1,:M2)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 0.60.

- Assume the host variable M1 is a DECIMAL(4,2) host variable with a value of 5.50, and host variable M2 is a DECIMAL(4,1) host variable with a value of 2.0.

```
SELECT MOD(:M1,:M2)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 1.50.

63. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN is returned with a warning. Otherwise, a division by zero warning or error is returned.

MONTH

The MONTH function returns the month part of a value.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.
- If *expression* is a number, it must be a date duration or timestamp duration. For the valid formats of datetime durations, see "Datetime operands and durations" on page 173.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a date, a timestamp, or a valid character-string representation of a date or timestamp:

  The result is the month part of the value, which is an integer between 1 and 12.
- If the argument is a date duration or timestamp duration:

  The result is the month part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example

- Select all rows from the EMPLOYEE table for people who were born (BIRTHDATE) in DECEMBER.

```
SELECT *
  FROM EMPLOYEE
  WHERE MONTH(BIRTHDATE) = 12
```

MONTHNAME

Returns a mixed case character string containing the name of the month (for example, January) for the month portion of the argument.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is VARCHAR(100). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the default CCSID of the current server.

Note

National language considerations: The name of the month that is returned is based on the language used for messages in the job. This name of the month is retrieved from message CPX3BC0 in message file QCPFMSG in library *LIBL.

Examples

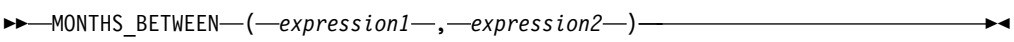
- Assume that the language used is US English.

```
SELECT MONTHNAME( '2003-01-02' )
  FROM SYSIBM.SYSDUMMY1
```

  Results in 'January'.

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns an estimate of the number of months between *expression1* and *expression2*.



expression1
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character-string, or a graphic-string.

If *expression1* is a character or graphic string, it must not be a CLOB or DBCLOB and its values must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83

expression2
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character-string, or a graphic-string.

If *expression2* is a character or graphic string, it must not be a CLOB or DBCLOB and its values must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83

If *expression1* represents a date that is later than *expression2*, then the result is positive. If *expression2* represents a date that is later than *expression1*, then the result is negative.

- If *expression1* and *expression2* represent dates with the same day of the month or the last day of the month, or both arguments represent the last day of their respective months, the result is a the whole number difference based on the year and month values ignoring any time portions of timestamp arguments.
- Otherwise, the whole number part of the result is the difference based on the year and month values. The fractional part of the result is calculated from the remainder based on an assumption that every month has 31 days. If either argument represents a timestamp, the arguments are effectively processed as timestamps with maximum precision, and the time portions of these values are also considered when determining the result.

The result of the function is a DECIMAL(31,15). If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples

- Calculate the months between two dates:

```
SELECT MONTHS_BETWEEN('2005-01-17', '2005-02-17')
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value -1.000000000000000

```
SELECT MONTHS_BETWEEN('2005-02-20', '2005-01-17')
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 1.096774193548387

- The following table contains additional examples:

Table 47. Additional examples using MONTHS_BETWEEN

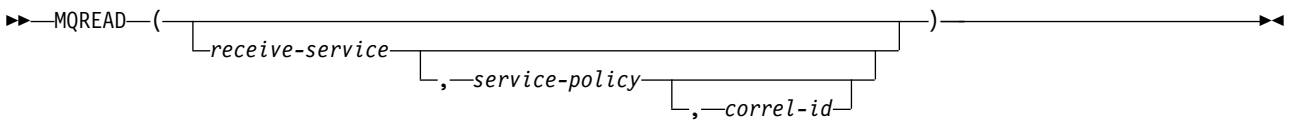
Value for argument *e1*	**Value for argument** *e2*	**Value returned by MONTHS_BETWEEN (***e1,e2***)**	**Value returned by DECIMAL(ROUND(MONTHS_BETWEEN(** *e1,e2***)*31,2), 15, 2)**	**Comment**
2005-02-02	2005-01-01	1.032258064516129	32.00	
2007-11-01-09.00.00.00000	2007-12-07-14.30.12.12345	-1.200945386592741	-37.23	
2007-12-13-09.40.30.00000	2007-11-13-08.40.30.00000	1.000000000000000	31.00	See Note 1
2007-03-15	2007-02-20	0.838709677419354	26.00	See Note 2
2008-02-29	2008-02-28-12.00.00	0.016129032258064	0.50	
2008-03-29	2008-02-29	1.000000000000000	31.00	
2008-03-30	2008-02-29	1.032258064516129	32.00	
2008-03-31	2008-02-29	1.000000000000000	31.00	See Note 3

Note:

1. The time difference is ignored because the day of the month is the same for both values.
2. The result is not 23 because, even though February has 28 days, the assumption is that all months have 31 days.
3. The result is not 33 because both dates are the last day of their respective month, and so the result is only based on the year and month portions.

MQREAD

The MQREAD function returns a message from a specified MQSeries location (return value of VARCHAR) without removing the message from the queue.



The MQREAD function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the message from the queue that is associated with *receive-service*, but instead returns the message at the beginning of the queue.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *receive-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. The first message with a matching correlation identifier is returned. For more information about MQSeries Application Messaging, see SQL Programming.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the identical *correl-id* must be specified to be recognized as a match. For example,

specifying a value of 'test' for *correl-id* on MQRECEIVE does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified or is an empty string or the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a varying-length string with a length attribute of 32000. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example reads the message at the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREAD ()
 FROM SYSIBM.SYSDUMMY1
```

- This example reads the message at the head of the queue specified by the service "MYSERVICE" using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREAD ('MYSERVICE')
 FROM SYSIBM.SYSDUMMY1
```

- This example reads the message at the head of the queue specified by the service "MYSERVICE", and using the policy "MYPOLICY".

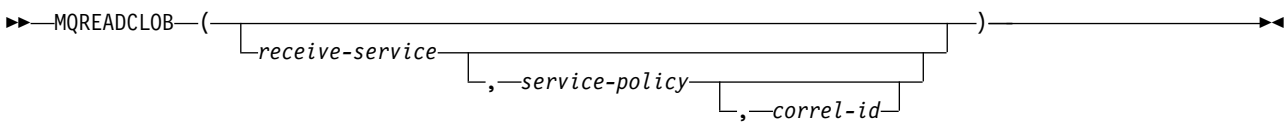
```
SELECT MQREAD ('MYSERVICE','MYPOLICY')
 FROM SYSIBM.SYSDUMMY1
```

- This example reads the first message with a correlation id that matches '1234' from the head of the queue specified by the service "MYSERVICE" using the policy "MYPOLICY".

```
SELECT MQREAD ('MYSERVICE','MYPOLICY','1234')
 FROM SYSIBM.SYSDUMMY1
```

MQREADCLOB

The MQREADCLOB function returns a message from a specified MQSeries location (return value of CLOB) without removing the message from the queue.



The MQREADCLOB function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the message from the queue that is associated with *receive-service*, but instead returns the message at the beginning of the queue.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *receive-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. The first message with a matching correlation identifier is returned. For more information about MQSeries Application Messaging, see SQL Programming.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the identical *correl-id* must be specified to be recognized as a match. For example,

specifying a value of 'test' for *correl-id* on MQRECEIVE does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified or is an empty string or the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a CLOB with a length attribute of 2 MB. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example reads the message at the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREADCLOB ()
 FROM SYSIBM.SYSDUMMY1
```

- This example reads the message at the head of the queue specified by the service "MYSERVICE" using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREADCLOB ('MYSERVICE')
 FROM SYSIBM.SYSDUMMY1
```

- This example reads the message at the head of the queue specified by the service "MYSERVICE", and using the policy "MYPOLICY".

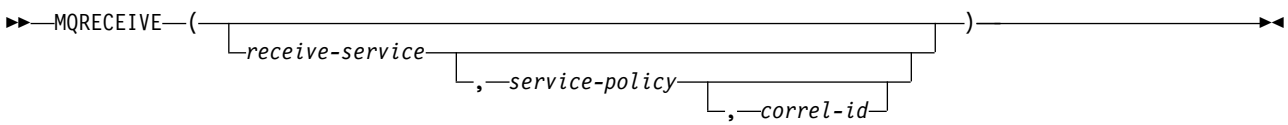
```
SELECT MQREADCLOB ('MYSERVICE','MYPOLICY')
 FROM SYSIBM.SYSDUMMY1
```

- This example reads the first message with a correlation id that matches '1234' from the head of the queue specified by the service "MYSERVICE" using the policy "MYPOLICY".

```
SELECT MQREADCLOB ('MYSERVICE','MYPOLICY','1234')
 FROM SYSIBM.SYSDUMMY1
```

MQRECEIVE

The MQRECEIVE function returns a message from a specified MQSeries location (return value of VARCHAR) with removal of the message from the queue.



The MQRECEIVE function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the message from the beginning of the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *receive-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. The first message with a matching correlation identifier is returned. For more information about MQSeries Application Messaging, see SQL Programming.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the identical *correl-id* must be specified to be recognized as a match. For example,

specifying a value of 'test' for *correl-id* on MQRECEIVE does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified or is an empty string or the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a varying-length string with a length attribute of 32000. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example receives the message at the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVE ()
 FROM SYSIBM.SYSDUMMY1
```

- This example receives the message at the head of the queue specified by the service "MYSERVICE" using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVE ('MYSERVICE')
 FROM SYSIBM.SYSDUMMY1
```

- This example receives the message at the head of the queue specified by the service "MYSERVICE" using the policy "MYPOLICY".

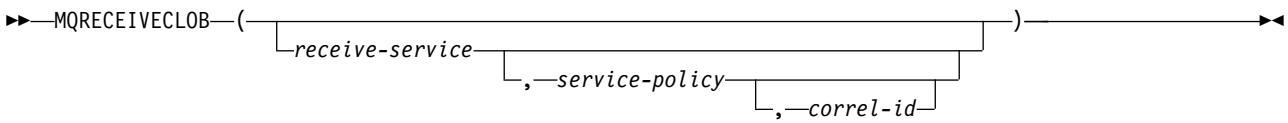
```
SELECT MQRECEIVE ('MYSERVICE','MYPOLICY')
 FROM SYSIBM.SYSDUMMY1
```

- This example receives the first message with a correlation id that matches '1234' from the head of the queue specified by the service "MYSERVICE" using the policy "MYPOLICY".

```
SELECT MQRECEIVE ('MYSERVICE','MYPOLICY','1234')
 FROM SYSIBM.SYSDUMMY1
```

MQRECEIVECLOB

The MQRECEIVECLOB function returns a message from a specified MQSeries location (return value of CLOB) with removal of the message from the queue.



The MQRECEIVE function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the message from the beginning of the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *receive-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. The first message with a matching correlation identifier is returned. For more information about MQSeries Application Messaging, see SQL Programming.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the identical *correl-id* must be specified to be recognized as a match. For example,

specifying a value of 'test' for *correl-id* on MQRECEIVE does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified or is an empty string or the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a CLOB with a length attribute of 2 MB. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example receives the message at the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVECLOB ()
 FROM SYSIBM.SYSDUMMY1
```

- This example receives the message at the head of the queue specified by the service "MYSERVICE" using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVECLOB ('MYSERVICE')
 FROM SYSIBM.SYSDUMMY1
```

- This example receives the message at the head of the queue specified by the service "MYSERVICE" using the policy "MYPOLICY".

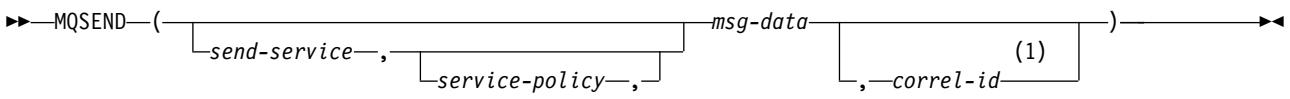
```
SELECT MQRECEIVECLOB ('MYSERVICE','MYPOLICY')
 FROM SYSIBM.SYSDUMMY1
```

- This example receives the first message with a correlation id that matches '1234' from the head of the queue specified by the service "MYSERVICE" using the policy "MYPOLICY".

```
SELECT MQRECEIVECLOB ('MYSERVICE','MYPOLICY','1234')
 FROM SYSIBM.SYSDUMMY1
```

MQSEND

The MQSEND function sends a message to a specified MQSeries location.



Notes:

1 The *correl-id* cannot be specified unless a *send-service* and a *service-policy* are also specified.

The MQSEND function sends the message data that is contained in *msg-data* to the MQSeries location that is specified by *send-service*, using the quality-of-service policy that is defined in *service-policy*. The message is sent using the MQSeries built-in format MQFMT_STRING.

send-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *send-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

msg-data
An expression that returns a value that is a built-in character string data type. If the expression is a CLOB, the value must not be longer than 2 MB. Otherwise, the value must not be longer than 32000 bytes. The value of the expression is the message data that is to be sent via MQSeries. A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation

identifier is often specified in request-and-reply scenarios to associate requests with replies. For more information about MQSeries Application Messaging, see SQL Programming.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQRECEIVE, the identical *correl-id* must be specified to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQSEND does not match a *correl-id* value of 'test ' (with trailing blanks) specified subsequently on an MQRECEIVE request.

If *correl-id* is not specified or is an empty string or the null value, a correlation identifier is not sent.

The result of the function is a varying-length string with a length attribute of 1. The result is nullable, even though a null value is never returned. The result is '1' if the function was successful or '0' if unsuccessful.

The CCSID of the result is the default SBCS CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example sends the string "Testing 123" to the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY), with no correlation identifier.

```
SELECT MQSEND ('Testing 123')
 FROM SYSIBM.SYSDUMMY1
```

- This example sends the string "Testing 345" to the service "MYSERVICE", using the policy "MYPOLICY", with no correlation identifier.

```
SELECT MQSEND ('MYSERVICE','MYPOLICY','Testing 345')
 FROM SYSIBM.SYSDUMMY1
```

- This example sends the string "Testing 678" to the service "MYSERVICE", using the policy "MYPOLICY", with correlation identifier "TEST3".

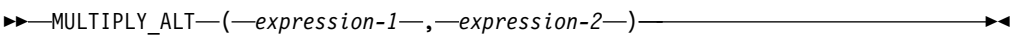
```
SELECT MQSEND ('MYSERVICE','MYPOLICY','Testing 678','TEST3')
 FROM SYSIBM.SYSDUMMY1
```

- This example sends the string "Testing 901" to the service "MYSERVICE", using the default policy (DB2.DEFAULT.POLICY), and no correlation identifier.

```
SELECT MQSEND ('MYSERVICE','Testing 901')
 FROM SYSIBM.SYSDUMMY1
```

MULTIPLY_ALT

The MULTIPLY_ALT scalar function returns the product of the two arguments as a decimal value. It is provided as an alternative to the multiplication operator, especially when the sum of the precisions of the arguments exceeds 63.



expression-1
An expression that returns a value of any built-in numeric data type (other than floating-point or decimal floating-point), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

expression-2
An expression that returns a value of any built-in numeric data type (other than floating-point or decimal floating-point), character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. *expression-2* cannot be zero.

The result of the function is a DECIMAL. The precision and scale of the result are determined as follows, using the symbols *p* and *s* to denote the precision and scale of the first argument, and the symbols *p'* and *s'* to denote the precision and scale of the second argument.

- The precision is MIN(*mp*, p+p')
- The scale is:
  - 0 if the scale of both arguments is 0
  - MIN(*ms*, s+s') if p+p' is less than or equal to *mp*
  - MIN(*ms*, MAX(MIN(3, s+s'), *mp*-(p-s+p'-s'))) if p+p' is greater than *mp*.

For information about the values of *p*, *s*, *ms*, and *mp*, see "Decimal arithmetic in SQL" on page 167.

If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The MULTIPLY_ALT function is a better choice than the multiplication operator when performing decimal arithmetic where a scale of at least 3 is wanted and the sum of the precisions exceeds 63. In these cases, the internal computation is performed so that overflows are avoided and then assigned to the result type value using truncation for any loss of scale in the final result. Note that the possibility of overflow of the final result is still possible when the scale is 3.

The following table compares the result types using MULTIPLY_ALT and the multiplication operator when the maximum precision is 31 and the maximum scale is 31:

Type of Argument 1	Type of Argument 2	Result using MULTIPLY_ALT	Result using multiplication operator
DECIMAL(31,3)	DECIMAL(15,8)	DECIMAL(31,3)	DECIMAL(31,11)

Type of Argument 1	Type of Argument 2	Result using MULTIPLY_ALT	Result using multiplication operator
DECIMAL(26,23)	DECIMAL(10,1)	DECIMAL(31,19)	DECIMAL(31,24)
DECIMAL(18,17)	DECIMAL(20,19)	DECIMAL(31,29)	DECIMAL(31,31)
DECIMAL(16,3)	DECIMAL(17,8)	DECIMAL(31,9)	DECIMAL(31,11)
DECIMAL(26,5)	DECIMAL(11,0)	DECIMAL(31,3)	DECIMAL(31,5)
DECIMAL(21,1)	DECIMAL(15,1)	DECIMAL(31,2)	DECIMAL(31,2)

Examples

- Multiply two values where the data type of the first argument is DECIMAL(26,3) and the data type of the second argument is DECIMAL(9,8). The data type of the result is DECIMAL(31,7).

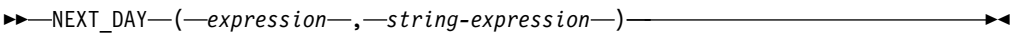
```
SELECT MULTIPLY_ALT(9876543210987654321098 7.654,5.43210987)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 53650467857887529485788 7.5277415.

  Note that the complete product of these two numbers is 53650467857887529485788 7.52774154498, but the last 4 digits are truncated to match the scale of the result data type. Using the multiplication operator with the same values will cause an arithmetic overflow, since the result data type is DECIMAL(31,11) and the result value has 24 digits left of the decimal, but the result data type only supports 20 digits.

NEXT_DAY

The NEXT_DAY function returns a date or timestamp value that represents the first weekday, named by *string-expression*, that is later than the date *expression*.



expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

string-expression
An expression that returns a built-in character string data type or graphic string data type that is not a CLOB or DBCLOB. The value must compare equal to the full name of a day of the week or compare equal to the abbreviation of a day of the week. For example, in the English language:

Day of Week	Abbreviation
MONDAY	MON
TUESDAY	TUE
WEDNESDAY	WED
THURSDAY	THU
FRIDAY	FRI
SATURDAY	SAT
SUNDAY	SUN

The minimum length of the input value is the length of the abbreviation. Leading and trailing blanks are trimmed from *string-expression*. The resulting value is then folded to uppercase, so the characters in the value may be in any case.

The result of the function has the same data type as expression, unless expression is a string, in which case the result data type is TIMESTAMP. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Any hours, minutes, seconds or fractional seconds information included in *expression* is not changed by the function. If *expression* is a string representing a date, the time information in the resulting TIMESTAMP value is all set to zero.

Note

National language considerations: The values of the days of the week (or abbreviations) in *string-expression* may either be the US English values listed in the table above or the values based on the language used for messages in the job. The non-abbreviated name of the day is retrieved from message CPX9034 in message file QCPFMSG in library *LIBL. The abbreviated name of the day is retrieved from message CPX9039 in message file QCPFMSG in library *LIBL.

Applications that need to run in many different language environments may want to consider using US English values since they will always be accepted in the NEXT_DAY function.

Example

- Assuming that the default language for the job is US English, set the host variable NEXTDAY with the date of the Tuesday following April 24, 2000.

  ```
  SET :NEXTDAY = NEXT_DAY(CURRENT_DATE, 'TUESDAY')
  ```

  The host variable NEXTDAY is set with the value of '2000–04–25', assuming that the value of the CURRENT_DATE special register is '2000–04–24'.

- Assuming that the default language for the job is US English, set the host variable NEXTDAY with the date of the first Monday in May, 2000. Assume the host variable DAYHV = 'MON'.

  ```
  SET :NEXTDAY = NEXT_DAY(LAST_DAY(CURRENT_TIMESTAMP), :DAYHV)
  ```

  The host variable NEXTDAY is set with the value of '2000-05-01-12.01.01.123456', assuming that the value of the CURRENT_TIMESTAMP special register is '2000-04-24-12.01.01.123456'.

- Assuming that the default language for the job is US English,

  ```
  SELECT NEXT_DAY('2000-04-24', 'TUESDAY')
  FROM SYSIBM.SYSDUMMY1
  ```

  Returns '2000-04-25-00.00.00.000000', which is the Tuesday following '2000-04-24'.

NORMALIZE_DECFLOAT

The NORMALIZE_DECFLOAT function returns a DECFLOAT value equal to the input argument set to its simplest form.

The NORMALIZE_DECFLOAT function returns a decimal floating-point value equal to the input argument set to its simplest form; that is, a non-zero number with trailing zeros in the coefficient has those zeros removed. This may require representing the number in normalized form by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly. A zero value has its exponent set to 0.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If the data type of the argument is SMALLINT, INTEGER, REAL, DOUBLE, DECIMAL(p,s) where p <=16, or NUMERIC(p,s) where p <=16, then the argument is converted to DECFLOAT(16) for processing. Otherwise, the argument is converted to DECFLOAT(34) for processing.

If the argument is a special value then the general rules for arithmetic operations apply. See "General arithmetic operation rules for DECFLOAT" on page 169 for more information.

The result of the function is a DECFLOAT(16) value if the data type of *expression* after conversion to decimal floating-point is DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

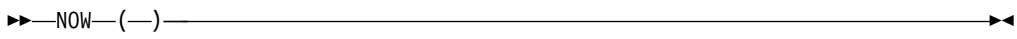
Examples

The following examples show the result of using the NORMALIZE_DECFLOAT function on various DECFLOAT values:

```
NORMALIZE_DECFLOAT(DECFLOAT(2.1))          =   2.1
NORMALIZE_DECFLOAT(DECFLOAT(-2.0))         =  -2
NORMALIZE_DECFLOAT(DECFLOAT(1.200))        =   1.2
NORMALIZE_DECFLOAT(DECFLOAT(-120))         =  -1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(120.00))       =   1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(0.00))         =   0
NORMALIZE_DECFLOAT(-NAN)                   =   -NAN
NORMALIZE_DECFLOAT(-INFINITY)              =   -INFINITY
```

NOW

The NOW function returns a timestamp based on a reading of the time-of-day clock when the SQL statement is executed at the current server. The value returned by the NOW function is the same as the value returned by the CURRENT_TIMESTAMP special register. If this function is used more than once within a single SQL statement, or used with the CURDATE or CURTIME scalar functions or the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP special registers within a single statement, all values are based on a single clock reading.



The data type of the result is a timestamp. The result cannot be null.

Note

Syntax alternatives: The CURRENT_TIMESTAMP special register should be used for maximal portability. For more information, see "Special registers" on page 129.

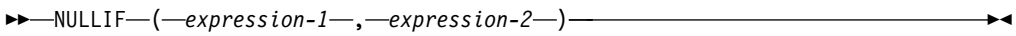
Example

- Return the current timestamp based on the time-of-day clock.

```
SELECT NOW()
  FROM SYSIBM.SYSDUMMY1
```

NULLIF

The NULLIF function returns a null value if the arguments compare equal, otherwise it returns the value of the first argument.



The arguments must be compatible data types.

expression-1
An expression that returns a value of any built-in data type other than a DATALINK or XML or any distinct data type other than a distinct type that is based on a DATALINK or XML.

expression-2
An expression that returns a value of any built-in data type other than a DATALINK or XML or any distinct data type other than a distinct type that is based on a DATALINK or XML .

The attributes of the result are the attributes of the first argument. The result can be null. The result is null if the first argument is null or if both arguments are equal.

The result of using NULLIF(e1,e2) is the same as using the expression

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

Note that when e1=e2 evaluates to unknown (because one or both arguments is NULL), CASE expressions consider this not true. Therefore, in this situation, NULLIF returns the value of the first operand, e1.

Example

- Assume host variables PROFIT, CASH, and LOSSES have DECIMAL data types with the values 4500.00, 500.00, and 5000.00 respectively:

  ```
  SELECT NULLIF (:PROFIT + :CASH, :LOSSES )
    FROM SYSIBM.SYSDUMMY1
  ```

  Returns the null value.

OCTET_LENGTH

The OCTET_LENGTH function returns the length of a string expression in octets (bytes).

See "LENGTH" on page 392 and "CHARACTER_LENGTH" on page 285 for similar functions.

expression
An expression that returns a value of any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The result of the function is DECIMAL(31). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is the number of octets (bytes) in the argument. The length of a string includes trailing blanks. The length of a varying-length string is the actual length in octets (bytes), not the maximum length.

Example

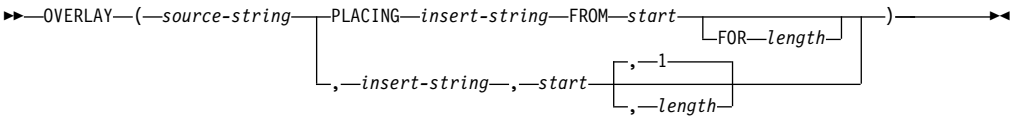
- Assume table T1 has a GRAPHIC(10) column called C1.

```
SELECT OCTET_LENGTH( C1 )
  FROM T1
```

  Returns the value 20.

OVERLAY

Returns a string where *length* characters have been deleted from *source-string* beginning at *start* and where *insert-string* has been inserted into *source-string* beginning at *start*.



source-string
An expression that specifies the source string. The *source-string* may be any built-in numeric, string, or datetime expression. It must be compatible with the *insert-string*. For more information about data type compatibility, see "Assignments and comparisons" on page 98. A numeric or datetime argument is cast to VARCHAR with a CCSID that is the default SBCS CCSID at the current server before evaluating the function. For more information about converting numeric or datetime to a varying character string, see "VARCHAR" on page 531. The actual length of the string must be greater than zero.

The OVERLAY function is identical to the INSERT function except that the arguments are in a different order and *length* is optional.

insert-string
An expression that specifies the string to be inserted into *source-string*, starting at the position identified by *start*. The *insert-string* may be any built-in numeric, string, or datetime expression. It must be compatible with the *source-string*. For more information about data type compatibility, see "Assignments and comparisons" on page 98. A numeric or datetime argument is cast to VARCHAR with a CCSID that is the default SBCS CCSID at the current server before evaluating the function. For more information about converting numeric or datetime to a varying character string, see "VARCHAR" on page 531.

start
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer specifies the starting character within *source-string* where the deletion of characters and the insertion of another string is to begin. The value of the integer must be in the range of 1 to the length of *source-string* plus one.

length
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer specifies the number of characters that are to be deleted from *source-string*, starting at the character position identified by *start*. The value of the integer must be in the range of 0 to the length of *source-string*.

The data type of the result of the function depends on the data type of the first and second arguments. The result data type is the same as if the two arguments were concatenated except that the result is always a varying-length string. For more information see "Conversion rules for operations that combine strings" on page 120.

The length attribute of the result depends on the arguments:

- If *start* and *length* are constants, the length attribute of the result is:

```
L1 - MIN((L1-V2 + 1), V3) + L4
```

where:

```
L1 is the length attribute of source-string
V2 depends on the encoding schema of source-string:
- If the source-string is UTF-8, the value MIN(L1+1,start*3)
- If the source-string is mixed data, the value MIN(L1+1,(start-1)*2.5+4)
- Otherwise, the value of start
V3 is the value of length
L4 is the length attribute of insert-string
```

- Otherwise, the length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string*.

If the length attribute of the result exceeds the maximum for the result data type, an error is returned.

The actual length of the result is:

```
A1 - MIN((A1 -V2 + 1), V3) + A4
```

where:

```
A1 is the actual length of source-string
V2 is the value of start
V3 is the value of length
A4 is the actual length of insert-string
```

If the actual length of the result string exceeds the maximum for the result data type, an error is returned.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is determined by the CCSID of *source-string* and *insert-string*. The resulting CCSID is the same as if the two arguments were concatenated. For more information, see "Conversion rules for operations that combine strings" on page 120.

Examples

- The following example shows how the string 'INSERTING' can be changed into other strings. The use of the CHAR function limits the length of the resulting string to 10 characters.

```
SELECT OVERLAY('INSERTING', 'IS', 4, 2),
       OVERLAY('INSERTING', 'IS', 4, 0),
       OVERLAY('INSERTING', '', 4, 2)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 'INSISTING ', 'INSISERTING', and 'INSTING '.

- The previous example demonstrated how to insert text into the middle of some text. This example shows how to insert text before some text by using 1 as the starting point (*start*).

```
SELECT OVERLAY('INSERTING', 'XX', 1, 0),
       OVERLAY('INSERTING', 'XX', 1, 1),
       OVERLAY('INSERTING', 'XX', 1, 2),
       OVERLAY('INSERTING', 'XX', 1, 3)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 'XXINSERTING', 'XXNSERTING', 'XXSERTING ', and 'XXERTING '.

- The following example shows how to insert text after some text. Add 'XX' at the end of string 'ABCABC'. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT OVERLAY('ABCABC', 'XX', 7, 0)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 'ABCABCXX '.

- The following example changes the string 'Hegelstraße' to 'Hegelstrasse'.

```
SELECT OVERLAY('Hegelstraße', 'ss', 10, 1)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 'Hegelstrasse'.

- Assume the variable UTF8_VAR is defined as UTF8 and UTF16_VAR is defined as UTF16. Assume both contain Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character.

```
SELECT OVERLAY(UTF8_VAR, 'C', 1),
       OVERLAY(UTF8_VAR, 'C', 5),
       OVERLAY(UTF16_VAR, 'C', 1),
       OVERLAY(UTF16_VAR, 'C', 5)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 'CN~AB', '&N~AC', 'CN~AB', and '&N~AC'.

PI

Returns the value of π 3.141592653589793. There are no arguments.

The result of the function is double-precision floating-point. The result cannot be null.

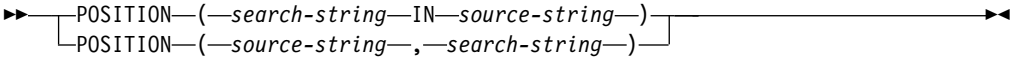
Example

- The following returns the circumference of a circle with diameter 10:

```
SELECT PI()*10
  FROM SYSIBM.SYSDUMMY1
```

POSITION

The POSITION function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*.



See the related functions "LOCATE" on page 396 and "POSSTR" on page 444.

search-string
An expression that specifies the string that is to be searched for. *search-string* can be any built-in numeric, datetime, or string expression. It must be compatible with the *source-string*. A numeric or datetime argument is cast to a character string before evaluating the function. For more information about converting numeric and datetime to a character string, see "VARCHAR" on page 531.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* can be any built-in numeric, datetime, or string expression. A numeric or datetime argument is cast to a character string before evaluating the function. For more information about converting numeric and datetime to a character string, see "VARCHAR" on page 531.

The result of the function is a large integer. If either of the arguments can be null, the result can be null. If either of the arguments is null, the result is the null value.

The POSITION function operates on a character basis. Because POSITION operates on a character-string basis, any shift-in and shift-out characters are not required to be in exactly the same position, and their only significance is to indicate which characters are SBCS and which characters are DBCS.

If the CCSID of the *search-string* is different from the CCSID of the *source-string*, it is converted to the CCSID of the *source-string*. If the CCSID of the *source-string* is mixed data or UTF-8, CCSID conversion to UTF-16 will occur.

If a collating sequence other than *HEX is in effect when the statement that contains the POSITION function is executed and the arguments are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the collating sequence. An ICU collating sequence table cannot be specified with the POSITION function.

If the *search-string* has a length of zero, the result returned by the function is 1. Otherwise:

- if the *source-string* has a length of zero, the result returned by the function is 0.
- Otherwise,
  - If the value of *search-string* is equal to an identical length of substring of contiguous positions within the value of *source-string*, then the result returned by the function is the starting position of the first such substring within the *source-string* value.

– Otherwise, the result returned by the function is 0.[64]

Example

- Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD' within the NOTE_TEXT column for all entries in the IN_TRAY table that contain these words.

```
SELECT RECEIVED, SUBJECT, POSITION('GOOD', NOTE_TEXT)
  FROM IN_TRAY
  WHERE POSITION(NOTE_TEXT, 'GOOD') <> 0
```

- Assume that NOTE is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen lives on Hegelstraße'. Find the character position of the character 'ß' in the string.

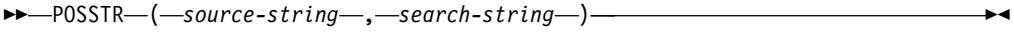
```
SELECT POSITION( NOTE, 'ß'), POSSTR( NOTE, 'ß')
  FROM T1
```

  Returns the value 26 for POSITION and 27 for POSSTR.

64. This includes the case where the *search-string* is longer than the *source-string*.

POSSTR

The POSSTR function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*.



See the related functions, "LOCATE" on page 396 and "POSITION" on page 442.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* may be any built-in numeric, datetime, or string expression. A numeric or datetime argument is cast to a character string before evaluating the function. For more information about converting numeric and datetime to a character string, see "VARCHAR" on page 531.

search-string
An expression that specifies the string that is to be searched for. *search-string* may be any built-in numeric, datetime, or string expression. It must be compatible with the *source-string*. A numeric or datetime argument is cast to a character string before evaluating the function. For more information about converting numeric and datetime to a character string, see "VARCHAR" on page 531.

The result of the function is a large integer. If either of the arguments can be null, the result can be null. If either of the arguments is null, the result is the null value.

The POSSTR function accepts mixed data strings. However, POSSTR operates on a strict byte-count basis without regard to single-byte or double-byte characters.[65] It is recommended that if either the *search-string* or *source-string* contains mixed data, POSITION should be used instead of POSSTR. The POSITION function operates on a character basis. In an EBCDIC encoding scheme, any shift-in and shift-out characters are not required to be in exactly the same position, and their only significance is to indicate which characters are SBCS and which characters are DBCS.

If the CCSID of the *search-string* is different than the CCSID of the *source-string*, it is converted to the CCSID of the *source-string*. If the CCSID of the *source-string* is mixed data or UTF-8, CCSID conversion to UTF-16 will occur.

If a collating sequence other than *HEX is in effect when the statement that contains the POSSTR function is executed and the arguments are SBCS data, mixed data, or Unicode data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the collating sequence. An ICU collating sequence table cannot be specified with the POSSTR function.

If the *search-string* has a length of zero, the result returned by the function is 1. Otherwise:

- if the *source-string* has a length of zero, the result returned by the function is 0.
- Otherwise,

65. For example, in an EBCDIC encoding scheme, if the *source-string* contains mixed data, the *search-string* will only be found if any shift-in and shift-out characters are also found in the *source-string* in exactly the same positions.

- If the value of *search-string* is equal to an identical length of substring of contiguous positions within the value of *source-string*, then the result returned by the function is the starting position of the first such substring within the *source-string* value.
- Otherwise, the result returned by the function is 0.[66]

Example

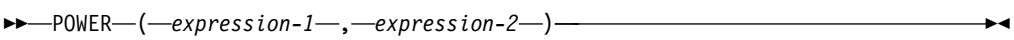
- Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD' within the NOTE_TEXT column for all entries in the IN_TRAY table that contain these words.

```
SELECT RECEIVED, SUBJECT, POSSTR(NOTE_TEXT, 'GOOD')
  FROM IN_TRAY
  WHERE POSSTR(NOTE_TEXT, 'GOOD') <> 0
```

66. This includes the case where the *search-string* is longer than the *source-string*.

POWER

The POWER® function returns the result of raising the first argument to the power of the second argument.



expression-1
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type.[67] A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

expression-2
An expression that returns a value of any built-in numeric data type. If the value of *expression-1* is equal to zero, then *expression-2* must be greater than or equal to zero. If the value of *expression-1* is less than zero, then *expression-2* must be an integer value.

If the data type of the argument is decimal floating-point, the data type of the result is DECFLOAT(34). Otherwise, the result of the function is a double-precision floating-point number. If both arguments are 0, the result is 1. If an argument can be null, the result can be null; if an argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: If either argument is decimal floating-point, both arguments are converted to DECFLOAT(34). For decimal floating-point values the special values are treated as follows:

- If either argument is NaN or -NaN, NaN is returned.[68]
- POWER(Infinity, any valid second argument) returns Infinity.
- POWER(-Infinity, any valid odd integer value) returns -Infinity.
- POWER(-Infinity, any valid even integer value) returns Infinity.
- POWER(0,Infinity) returns 0.
- POWER(1,Infinity) returns 1.
- POWER(any number greater than 1,Infinity) returns Infinity.
- POWER(any number greater than 0 and less than 1,Infinity) returns 0.
- POWER(any number less than 0,Infinity) returns NaN. [68]
- If either argument is sNaN or -sNaN, a warning or error is returned. [68]

Example

- Assume the host variable HPOWER is an integer with value 3.

```
SELECT POWER(2,:HPOWER)
  FROM SYSIBM.SYSDUMMY1
```

Returns the value 8.

67. The result of the POWER function is exactly the same as the result of exponentiation: *expression-1* ** *expression-2*.

68. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN is returned with a warning

QUANTIZE

The QUANTIZE function returns a decimal floating-point value that is equal in value (except for any rounding) and sign to *expression-1* and which has an exponent set equal to the exponent in *expression-2*.

expression-1
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If the data type of the argument is not a DECFLOAT value, it is converted to DECFLOAT(34) for processing.

expression-2
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If the data type of the argument is not a DECFLOAT value, it is converted to DECFLOAT(34) for processing.

If one argument (after conversion) is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) argument is converted to DECFLOAT(34) before the function is processed.

The coefficient of the result is derived from that of *expression-1*. It is rounded if necessary (if the exponent is being increased), multiplied by a power of ten (if the exponent is being decreased), or remains unchanged (if the exponent is already equal to that of *expression-2*).

If necessary, the rounding mode is used by the QUANTIZE function. See "CURRENT DECFLOAT ROUNDING MODE" on page 133 for more information.

Unlike other arithmetic operations on the DECFLOAT data type, if the length of the coefficient after the quantize operation would be greater than the precision of the resulting DECFLOAT number, an error occurs. This guarantees that unless there is an error, the exponent of the result of a QUANTIZE function is always equal to that of *expression-2*.

The result of the function is a DECFLOAT(16) value if both arguments are DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. If either argument can be null, the result can be null; if either argument is null, the result is the null value

Note

Results involving DECFLOAT special values: Decimal floating-point special values are treated as follows:

- If either argument is NaN and the first argument is not -NaN, then NaN is returned.
- If either argument is sNaN, then a warning or error occurs. [69]
- If either argument is -NaN and the first argument is not NaN, then -NaN is returned.
- If either argument is -sNaN, then a warning or error occurs. [69]

69. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN is returned with a warning.

- If both arguments are Infinity (positive or negative), then Infinity (positive or negative) is returned.
- If one argument is Infinity (positive or negative) and the other argument is not Infinity (positive or negative), then NaN is returned. [69]

Examples

The following examples illustrate the value that is returned for the QUANTIZE function given the input DECFLOAT values:

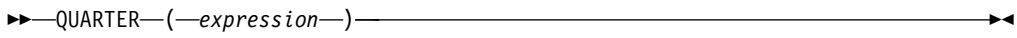
```
QUANTIZE(2.17, 0.001)          ==> 2.170
QUANTIZE(2.17, 0.01)           ==> 2.17
QUANTIZE(2.17, 0.1)            ==> 2.2
QUANTIZE(2.17, 1e+0)           ==> 2
QUANTIZE(2.17, 1e+1)           ==> 0E+1
QUANTIZE(2, Infinity)          ==> NaN (exception)
QUANTIZE(0, 1e+5)              ==> 0E+5
QUANTIZE(217, 1e-1)            ==> 217.0
QUANTIZE(217, 1e+0)            ==> 217
QUANTIZE(217, 1e+1)            ==> 2.2E+2
QUANTIZE(217, 1e+2)            ==> 2E+2
```

In the following example, the value -0 is returned for the QUANTIZE function. The CHAR function is used to avoid the potential removal of the minus sign by a client program.

```
CHAR(QUANTIZE(-0.1, 1))        ==> -0
```

QUARTER

The QUARTER function returns an integer between 1 and 4 that represents the quarter of the year in which the date resides. For example, any dates in January, February, or March will return the integer 1.



expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Using the PROJECT table, set the host variable QUART (INTEGER) to the quarter in which project 'PL2100' ended (PRENDATE).

```
SELECT QUARTER(PRENDATE)
  INTO :QUART
  FROM PROJECT
  WHERE PROJNO = 'PL2100'
```

  Results in QUART being set to 3.

RADIANS

The RADIANS function returns the number of radians for an argument that is expressed in degrees.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is cast to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

If the data type of the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

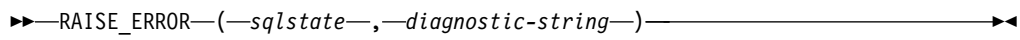
- Assume that host variable HDEG is an INTEGER with a value of 180. The following statement:

```
SELECT RADIANS(:HDEG)
  FROM SYSIBM.SYSDUMMY1
```

  Returns a double precision floating-point number with an approximate value of 3.1415926536.

RAISE_ERROR

The RAISE_ERROR function causes the statement that invokes the function to return an error with the specified SQLSTATE (along with SQLCODE -438) and diagnostic string.



sqlstate
An expression that returns a value of a built-in CHAR or VARCHAR data type with exactly 5 characters. The *sqlstate* value must follow the rules for application-defined SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or nonaccented uppercase letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00', '01', or '02' because these are not error classes.

If the SQLSTATE does not conform to these rules, an error is returned.

diagnostic-string
An expression that returns a value of a built-in CHAR or VARCHAR data type and a length up to 1000 bytes that describes the error condition. If the string is longer than 1000 bytes, it is truncated.

If an SQLCA is used, the following actions occur:

- The string is returned in the SQLERRMC field of the SQLCA.
- If the actual length of the string is longer than 70 bytes, it is truncated without a warning.

Since the data type of the result of RAISE_ERROR is undefined, it may only be used where parameter markers are allowed. To use this function in a context where parameter markers are not allowed (such as alone in a select list), you must use a cast specification to give a data type to the null value that is returned.

The RAISE_ERROR function always returns NULL with an undefined data type.

Example

- List employee numbers and education levels as Post Graduate, Graduate and Diploma. If an education level is greater than 20, raise an error.

```
SELECT EMPNO,
   CASE WHEN EDLEVEL < 16 THEN 'Diploma'
        WHEN EDLEVEL < 18 THEN 'Graduate'
        WHEN EDLEVEL < 21 THEN 'Post Graduate'
        ELSE RAISE_ERROR( '07001',
                 'EDLEVEL has a value greater than 20' )
   END
 FROM EMPLOYEE
```

RAND

The RAND function returns a floating point value greater than or equal to 0 and less than or equal to 1.

expression
If an expression is specified, it is used as the seed value. The argument must be an expression that returns a value of a built-in small integer, large integer, character-string, or graphic-string data type. A string argument is cast to integer before evaluating the function. For more information on converting strings to integer, see "INTEGER or INT" on page 384.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

A specific seed value will produce the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. If a seed value is not specified, a different sequence of random numbers is produced each time the query is executed.

The seed value is used only for the first invocation of an instance of the RAND function within a statement.

RAND is a non-deterministic function.

Example

- Assume that host variable HRAND is an INTEGER with a value of 100. The following statement:

```
SELECT RAND(:HRAND)
  FROM SYSIBM.SYSDUMMY1
```

  Returns a random floating-point number between 0 and 1, such as the approximate value .0121398.
- To generate values in a numeric interval other than 0 to 1, multiply the RAND function by the size of the wanted interval. For example, to get a random number between 0 and 10, such as the approximate value 5.8731398, multiply the function by 10:

```
SELECT RAND(:HRAND) * 10
  FROM SYSIBM.SYSDUMMY1
```

REAL

The REAL function returns a single-precision floating-point representation.

Numeric to Real

String to Real

The REAL function returns a single-precision floating-point representation of:

- A number
- A character or graphic string representation of a decimal number
- A character or graphic string representation of an integer
- A character or graphic string representation of a floating-point number
- A character or graphic string representation of a decimal floating-point number

Numeric to Real

numeric-expression
The argument is an expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable. If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned.

String to Real

string-expression
An expression that returns a value that is a character-string or graphic-string representation of a number.

If the argument is a *string-expression*, the result is the same number that would result from CAST(*string-expression* AS REAL). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant. If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned.

The single-byte character constant that must be used to delimit the decimal digits in *string-expression* from the whole part of the number is the default decimal point. For more information, see "Decimal point" on page 127.

The result of the function is a single-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

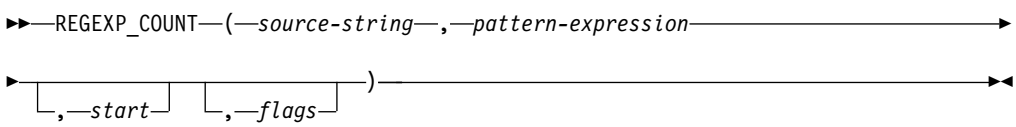
Example

- Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. To eliminate the possibility of out-of-range results, REAL is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, REAL(SALARY)/COMM
  FROM EMPLOYEE
  WHERE COMM > 0
```

REGEXP_COUNT

The REGEXP_COUNT function returns a count of the number of times that a regular expression pattern is matched in a string.



source-string
An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of a string must not be greater than 1 gigabyte.

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of the string must not be greater than 32K.

A valid *pattern-expression* consists of a set of characters and control characters that describe the pattern of the search. For a description of the valid control characters, see "Regular expression control characters" on page 220.

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be greater than or equal to 1. If the value of the integer is greater than the actual length of the *source-string*, the result is 0.

flags
An expression that specifies flags that control aspects of the pattern matching. The expression must return a value that is a built-in character string or graphic string data type. A character string with the FOR BIT DATA attribute or a binary string is not supported. The string can include one or more valid flag values and the combination of flag values must be valid. An empty string is the same as the value 'c'.

For a description of the valid flag characters, see "Regular expression flag values" on page 220.

The result of the function is an INTEGER representing the number of occurrences of the *pattern-expression* within the *source-string*. If the *pattern-expression* is not found and no argument is null, the result is 0.

If any argument of the REGEXP_COUNT function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

Prerequisites: In order to use the REGEXP_COUNT function, the International Components for Unicode (ICU) option must be installed

Processing: The regular expression processing is done using the International Components for Unicode (ICU) regular expression interface. For more information see, http://userguide.icu-project.org/strings/regexp.

If only three arguments are specified, the third argument may be a *start* or *flags* argument. If the third argument is a string, it is interpreted as a *flags* argument. Otherwise, it is interpreted as a *start* argument.

Syntax Alternatives: REGEXP_MATCH_COUNT is a synonym for REGEXP_COUNT.

Example

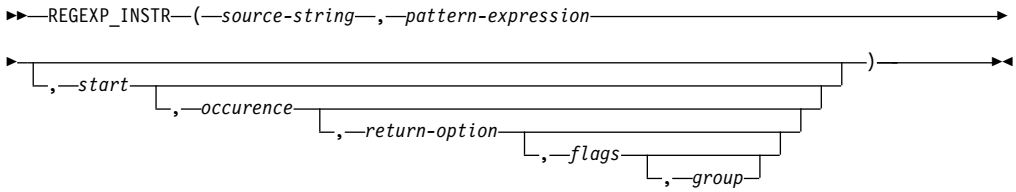
- Count the number of times "Steven" or "Stephen" occurs in the string "Steven Jones and Stephen Smith are the best players".

```
SELECT REGEXP_COUNT(
  'Steven Jones and Stephen Smith are the best players',
  'Ste(v|ph)en')
FROM sysibm.sysdummy1
```

  The result is 2.

REGEXP_INSTR

The REGEXP_INSTR returns the starting position or the position after the end of the matched substring, depending on the value of the *return_option* argument.



source-string
An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of a string must not be greater than 1 gigabyte.

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of the string must not be greater than 32K.

A valid *pattern-expression* consists of a set of characters and control characters that describe the pattern of the search. For a description of the valid control characters, see "Regular expression control characters" on page 220.

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be greater than or equal to 1. If the value of the integer is greater than the actual length of the *source-string*, the result is 0.

occurrence
An expression that specifies which occurrence of the *pattern-expression* within *source-string* to search for. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be greater than or equal to 1. If *occurrence* is not specified, the default value is 1 which indicates that only the first occurrence of *pattern-expression* is considered.

return-option
An expression that specifies whether to return the starting position or the position after the end of the string that matches the pattern. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be equal to 0 or 1. A value of 0

returns the starting position of the occurrence. A value of 1 returns the ending position of the occurrence. If *return-option* is not specified, the default value is 0.

flags
An expression that specifies flags that control aspects of the pattern matching. The expression must return a value that is a built-in character string or graphic string data type. A character string with the FOR BIT DATA attribute or a binary string is not supported. The string can include one or more valid flag values and the combination of flag values must be valid. An empty string is the same as the value 'c'.

For a description of the valid flag characters, see "Regular expression flag values" on page 220.

group
An expression that specifies which capture group of the *pattern-expression* is used to determine the position within *source-string* to return. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be greater than or equal to 0 and must not be greater than the number of capture groups in the *pattern-expression*. If *group* is not specified, the default is 0 which indicates the entire string that matches the entire pattern is returned.

The result of the function is a large integer. If the *pattern-expression* is found, the result is a number from 1 to *n*, where *n* is the actual length of the *source-string* plus 1. The result value represents the position used to process the function. If the *pattern-expression* is not found and no argument is null, the result is 0.

If any argument of the REGEXP_INSTR function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

Prerequisites: In order to use the REGEXP_INSTR function, the International Components for Unicode (ICU) option must be installed

Processing: The regular expression processing is done using the International Components for Unicode (ICU) regular expression interface. For more information see, http://userguide.icu-project.org/strings/regexp.

Examples

- *Example 1*: Find the first occurrence of a 'o' which has a character preceding it.

```
SELECT REGEXP_INSTR('hello to you', '.o',1,1)
FROM sysibm.sysdummy1
```

  The result is 4, which is the position of the second 'l' character.
- *Example 2*: Find the second occurrence of a 'o' which has a character preceding it.

```
SELECT REGEXP_INSTR('hello to you', '.o',1,2)
FROM sysibm.sysdummy1
```

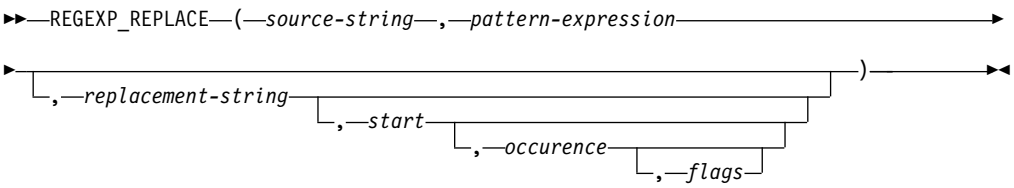
  The result is 7, which is the position of the character 't'.
- *Example 3*: Find the position after the third occurrence of the first capture group of the regular expression '(.o).' using case insensitive matching.

```
SELECT REGEXP_INSTR('hello to you', '(.o).', 1,3,1,'i',1)
FROM sysibm.sysdummy1
```

The result is 12, which is the position of the character 'u' at the end of the string.

REGEXP_REPLACE

The REGEXP_REPLACE function returns a modified version of the source string where occurrences of the regular expression pattern found in the source string are replaced with the specified replacement string.



source-string
An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of a string must not be greater than 1 gigabyte.

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of the string must not be greater than 32K.

A valid *pattern-expression* consists of a set of characters and control characters that describe the pattern of the search. For a description of the valid control characters, see "Regular expression control characters" on page 220.

replacement-string
An expression that specifies the replacement string for matching substrings. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of the string must not be greater than 32K. If *replacement-string* is not specified, the default is the empty string.

The content of the *replacement-string* can include references to capture group text from the search to use in the replacement text. These references are of the form '$n' or '\n'33, where n is the number of the capture group and 0 represents the entire string that matches the pattern. The value for n must be in the range 0 to 9 and not greater than the number of capture groups in the pattern. For example, either '$2' or '\2' can be used to refer to the content found in *source-string* for the second capture group specified in *pattern-expression*. If the *replacement-string* must include a literal reference to a '$' or '\' character, then the '\' character must precede the literal reference so that it appears in the *replacement-string* as '\$' or '\\'.

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to

INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be greater than or equal to 1. If the value of the integer is greater than the actual length of the *source-string*, the original string is returned.

occurrence
An expression that specifies which occurrence of the *pattern-expression* within *source-string* to search for and replace. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be greater than or equal to 0. If *occurrence* is not specified, the default value is 0 which indicates that all occurrences of *pattern-expression* in *source-string* are replaced.

flags
An expression that specifies flags that control aspects of the pattern matching. The expression must return a value that is a built-in character string or graphic string data type. A character string with the FOR BIT DATA attribute or a binary string is not supported. The string can include one or more valid flag values and the combination of flag values must be valid. An empty string is the same as the value 'c'.

For a description of the valid flag characters, see "Regular expression flag values" on page 220.

The result of the function is a string. If there are no occurrences of the pattern to be replaced and no argument is null, the original string is returned. The data type of the string is the same as if the first and third arguments were concatenated except that the result is always a varying-length string. For more information, see "With the concatenation operator" on page 170.

The length attribute of the result data type is determined based on the length attributes of the *source-string* and the *replacement-string* using the following calculation: MIN(MaxTypeLen, LAS+(LAS+1)*LAR) where MaxTypeLen is the maximum length attribute for the data type of the result, LAS is the length attribute for the data type of *source-string*, and LAR is the length attribute for the data type of *replacement-string*. If *replacement-string* is not specified, the value for LAR is 0. If the actual length of the result string exceeds the maximum for the return data type, an error is returned.

The CCSID of the result is determined by the CCSID of the *source-string* and the *replacement-string*. The resulting CCSID is the same as if the first and third arguments were concatenated. For more information, see "Rules for result data types" on page 115.

If any argument of the REGEXP_REPLACE function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

Prerequisites: In order to use the REGEXP_REPLACE function, the International Components for Unicode (ICU) option must be installed

Processing: The regular expression processing is done using the International Components for Unicode (ICU) regular expression interface. For more information see, http://userguide.icu-project.org/strings/regexp.

Example

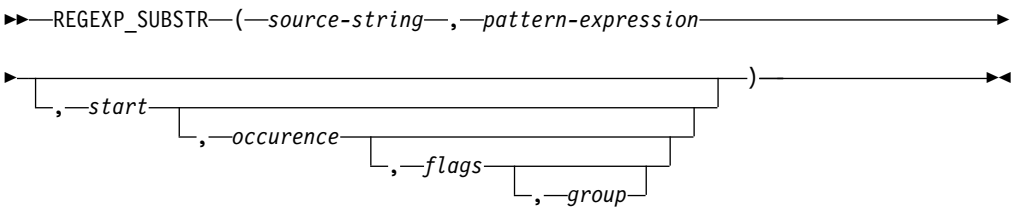
- Replace the second occurrence of the pattern 'R.d' with 'Orange' using a case sensitive search.

```
SELECT REGEXP_REPLACE(
  'Red Yellow RED Blue Red Green Blue',
  'R.d','Orange',1,2,'c')
FROM sysibm.sysdummy1
```

  The result is 'Red Yellow RED Blue Orange Green Blue'.

REGEXP_SUBSTR

The REGEXP_SUBSTR function returns one occurrence of a substring of a string that matches the regular expression pattern.



source-string
An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of a string must not be greater than 1 gigabyte.

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the value is not a UTF-16 DBCLOB, it is implicitly cast to a UTF-16 DBCLOB before searching for the regular expression pattern. A character string with the FOR BIT DATA attribute or a binary string is not supported. The length of the string must not be greater than 32K.

A valid *pattern-expression* consists of a set of characters and control characters that describe the pattern of the search. For a description of the valid control characters, see "Regular expression control characters" on page 220.

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be greater than or equal to 1. If the value of the integer is greater than the actual length of the *source-string*, the result is the null value.

occurrence
An expression that specifies which occurrence of the *pattern-expression* within *source-string* to search for. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see "INTEGER or INT" on page 384. The value of the integer must be greater than or equal to 1. If *occurrence* is not specified, the default value is 1 which indicates that only the first occurrence of *pattern-expression* is considered.

flags
An expression that specifies flags that control aspects of the pattern matching. The expression must return a value that is a built-in character string or graphic string data type. A character string with the FOR BIT DATA attribute or a

binary string is not supported. The string can include one or more valid flag values and the combination of flag values must be valid. An empty string is the same as the value 'c'.

For a description of the valid flag characters, see “Regular expression flag values” on page 220.

group
An expression that specifies which capture group of the *pattern-expression* within *source-string* to return. The expression must return a value of any built-in numeric, character-string, or graphic-string data type. The argument is cast to INTEGER before evaluating the function. For more information about converting to INTEGER, see “INTEGER or INT” on page 384. The value of the integer must be greater than or equal to 0 and must not be greater than the number of capture groups in the *pattern-expression*. If *group* is not specified, the default is 0 which indicates the entire string that matches the entire pattern is returned.

The result of the function is a string. The data type of the result depends on the data type of *source-string*:

Data type of *source-string*	**Data Type of the Result for REGEXP_SUBSTR**
CHAR or VARCHAR or numeric or datetime	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB

The length attribute of the result data type is same as the length attribute of the *source-string*. The actual length of the result is the length of the occurrence in the string that matches the *pattern-expression*. If the *pattern-expression* is not found, the result is the null value.

The CCSID of the result is the same as the *source-string*.

If any argument of the REGEXP_SUBSTR function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

Prerequisites: In order to use the REGEXP_SUBSTR function, the International Components for Unicode (ICU) option must be installed

Processing: The regular expression processing is done using the International Components for Unicode (ICU) regular expression interface. For more information see, http://userguide.icu-project.org/strings/regexp.

Syntax Alternatives: REGEXP_EXTRACT is a synonym for REGEXP_SUBSTR.

Examples

- *Example 1*: Return the string which matches any character preceding a 'o'.

```
SELECT REGEXP_SUBSTR('hello to you', '.o',1,1)
FROM sysibm.sysdummy1
```

The result is 'lo'.

- *Example 2*: Return the second string occurrence which matches any character preceding a 'o'.

```
SELECT REGEXP_SUBSTR('hello to you', '.o',1,2)
FROM sysibm.sysdummy1
```

The result is 'to'.

- *Example 3*: Return the third string occurrence which matches any character preceding a 'o'.

```
SELECT REGEXP_SUBSTR('hello to you', '.o',1,3)
FROM sysibm.sysdummy1
```

The result is 'yo'.

REPEAT

The REPEAT function returns a string composed of *expression* repeated *integer* times.

expression
An expression that specifies the string to be repeated. The string must be a built-in numeric or string expression. A numeric argument is cast to a character string before evaluating the function. For more information on converting numeric to a character string, see "VARCHAR" on page 531.

integer
An expression that returns a built-in BIGINT, INTEGER, or SMALLINT data type whose value is a positive integer or zero. The integer specifies the number of times to repeat the string.

The data type of the result of the function depends on the data type of the first argument:

Data type of *string-expression*	**Data type of the Result**
CHAR or VARCHAR or any numeric type	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

If *integer* is a constant, the length attribute of the result is the minimum of the length attribute of *string-expression* times *integer* and the maximum length of the result data type. Otherwise, the length attribute depends on the data type of the result:

- 1,048,576 for BLOB, CLOB, or DBCLOB
- 4000 for VARCHAR or VARBINARY
- 2000 for VARGRAPHIC

The actual length of the result is the actual length of *string-expression* times *integer*. If the actual length of the result string exceeds the maximum for the return type, an error is returned.

If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The CCSID of the result is the CCSID of *string-expression*.[70]

Examples

- Repeat 'abc' two times to create 'abcabc'.

70. If the value of *string-expression* is mixed data that is not a properly formed mixed data string, the result will not be a properly formed mixed data string.

```
SELECT REPEAT('abc', 2)
  FROM SYSIBM.SYSDUMMY1
```

- List the phrase 'REPEAT THIS' five times. Use the CHAR function to limit the output to 60 bytes.

```
SELECT CHAR( REPEAT('REPEAT THIS', 5), 60)
  FROM SYSIBM.SYSDUMMY1
```

  This example results in 'REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS '.

- For the following query, the LENGTH function returns a value of 0 because the result of repeating a string zero times is an empty string, which is a zero-length string.

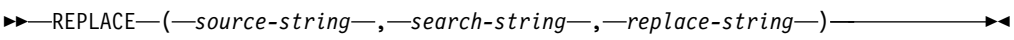
```
SELECT LENGTH( REPEAT('REPEAT THIS', 0) )
  FROM SYSIBM.SYSDUMMY1
```

- For the following query, the LENGTH function returns a value of 0 because the result of repeating an empty string any number of times is an empty string, which is a zero-length string.

```
SELECT LENGTH( REPEAT('', 5) )
  FROM SYSIBM.SYSDUMMY1
```

REPLACE

The REPLACE function replaces all occurrences of *search-string* in *source-string* with *replace-string*. If *search-string* is not found in *source-string*, *source-string* is returned unchanged.



source-string
An expression that specifies the source string. The *source-string* must be a built-in numeric or string expression. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

search-string
An expression that specifies the string to be removed from the source string. The *search-string* must be a built-in numeric or string expression. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

replace-string
An expression that specifies the replacement string. The *replace-string* must be a built-in numeric or string expression. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

source-string, *search-string*, and *replace-string* must be compatible. For more information about data type compatibility, see "Assignments and comparisons" on page 98.

The data type of the result of the function depends on the data type of the arguments. The result data type is the same as if the three arguments were concatenated except that the result is always a varying-length string. For more information see "Conversion rules for operations that combine strings" on page 120.

The length attribute of the result depends on the arguments:

- If *search-string* is variable length, the length attribute of the result is:
  ```
  (L3 * L1)
  ```
- If the length attribute of *replace-string* is less than or equal to the length attribute of *search-string*, the length attribute of the result is the length attribute of *source-string*
- Otherwise, the length attribute of the result is:
  ```
  (L3 * (L1/L2)) + MOD(L1,L2)
  ```

where:

```
L1 is the length attribute of source-string
L2 is the length attribute of search-string
L3 is the length attribute of replace-string
```

If the length attribute of the result exceeds the maximum for the result data type, an error is returned.

The actual length of the result is the actual length of *source-string* plus the number of occurrences of *search-string* that exist in *source-string* multiplied by the actual

length of *replace-string* minus the actual length of *search-string*. If the actual length of the result string exceeds the maximum for the result data type, an error is returned.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is determined by the CCSID of *source-string*, *search-string*, and *replace-string*. The resulting CCSID is the same as if the three arguments were concatenated. For more information, see "Conversion rules for operations that combine strings" on page 120.

Examples

- Replace all occurrences of the character 'N' in the string 'DINING' with 'VID'. Use the CHAR function to limit the output to 10 bytes.

```
SELECT CHAR(REPLACE( 'DINING', 'N', 'VID' ), 10)
  FROM SYSIBM.SYSDUMMY1
```

  The result is the string 'DIVIDIVIDG'.

- Replace string 'ABC' in the string 'ABCXYZ' with nothing, which is the same as removing 'ABC' from the string.

```
SELECT REPLACE( 'ABCXYZ', 'ABC', '' )
  FROM SYSIBM.SYSDUMMY1
```

  The result is the string 'XYZ'.

- Replace string 'ABC' in the string 'ABCCABCC' with 'AB'. This example illustrates that the result can still contain the string that is to be replaced (in this case, 'ABC') because all occurrences of the string to be replaced are identified prior to any replacement.

```
SELECT REPLACE( 'ABCCABCC', 'ABC', 'AB' )
  FROM SYSIBM.SYSDUMMY1
```

  The result is the string 'ABCABC'.

RID

The RID function returns the relative record number of a row as a BIGINT.

table-designator
A table designator that can be used to qualify a column in the same relative location in the SQL statement as the RID function. For more information about table designators, see "Table designators" on page 143.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

The *table-designator* must not identify a *table-function* or a *collection-derived-table*.

If the argument identifies a view, common table expression, or nested table expression, the function returns the relative record number of its base table. If the argument identifies a view, common table expression, or nested table expression derived from more than one base table, the function returns the relative record number of the first table in the outer subselect of the view, common table expression, or nested table expression.

If the argument identifies a distributed table, the function returns the relative record number of the row on the node where the row is located. If the argument identifies a partitioned table, the function returns the relative record number of the row in the partition where the row is located. This means that RID will not be unique for each row of a partitioned or distributed table.

The argument must not identify a view, common table expression, or nested table expression whose outer fullselect subselect includes an aggregate function, a GROUP BY clause, a HAVING clause, a UNION clause, an INTERSECT clause, DISTINCT clause, or VALUES clause. The RID function cannot be specified in a SELECT clause if the fullselect contains an aggregate function, a GROUP BY clause, a HAVING clause, or a VALUES clause. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is a big integer. The result can be null.

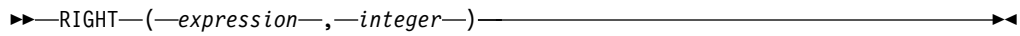
Example

- Return the relative record number and employee name from table EMPLOYEE for those employees in department 20.

```
SELECT RID(EMPLOYEE), LASTNAME
  FROM EMPLOYEE
  WHERE DEPTNO = 20
```

RIGHT

The RIGHT function returns the rightmost *integer* characters of *expression*.



If *expression* is a character string, the result is a character string. If *expression* is a graphic string, the result is a graphic string. If *expression* is a binary string, the result is a binary string.

expression
An expression that specifies the string from which the result is derived. The string must be a built-in numeric or string expression. A numeric argument is cast to a character string before evaluating the function. For more information on converting numeric to a character string, see "VARCHAR" on page 531.

A substring of *expression* is zero or more contiguous characters of *expression*. If *expression* is a character string or graphic string, a single character is either an SBCS, DBCS, or multiple-byte character. If *expression* is a binary string, the result is the number of bytes in the argument.

integer
An expression that returns a built-in integer data type. The integer specifies the length of the result. *integer* must be greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *expression*.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *expression* and a data type that depends on the data type of *expression*:

Data type of *expression*	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The actual length of the result is *integer*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of *expression*.

Example

- Assume that host variable ALPHA has a value of 'ABCDEF'. The following statement:

```
SELECT RIGHT( :ALPHA, 3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'DEF', which are the three rightmost characters in ALPHA.

- The following statement returns a zero length string.

```
SELECT RIGHT( 'ABCABC', 0)
  FROM SYSIBM.SYSDUMMY1
```

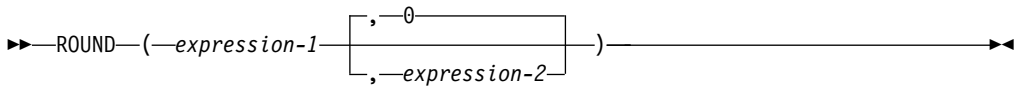
- Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'.

```
SELECT RIGHT(NAME, 5), SUBSTR(NAME, 3, 5)
  FROM T1
  WHERE NAME = 'Jürgen'
```

  Returns the value 'ürgen' for RIGHT and an unprintable string (X'BC7267656E') for SUBSTR(NAME, 3, 5).

ROUND

The ROUND function returns *expression–1* rounded to some number of places to the right or left of the decimal point.



expression–1
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is converted to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

If *expression-1* is a decimal floating-point data type, the DECFLOAT ROUNDING MODE will not be used. The rounding behavior of ROUND corresponds to a value of ROUND_HALF_UP. If a different rounding behavior is wanted, use the QUANTIZE function.

expression–2
An expression that returns a value of a built-in BIGINT, INTEGER, or SMALLINT data type.

If *expression–2* is positive, *expression–1* is rounded to the *expression–2* number of places to the right of the decimal point.

If *expression–2* is negative, *expression–1* is rounded to 1 + (the absolute value of *expression–2*) number of places to the left of the decimal point. If the absolute value of *expression–2* is greater than the number of digits to the left of the decimal point, the result is 0. (For example, ROUND(748.58,-4) returns 0.)

If *expression–2* is not specified, *expression–1* is rounded to zero places to the left of the decimal point.

If *expression–1* is positive, a digit value of 5 is rounded to the next higher positive number. If *expression–1* is negative, a digit value of 5 is rounded to the next lower negative number.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument, except that precision is increased by one if *expression–1* is DECIMAL or NUMERIC and the precision is less than the maximum precision (*mp*). For example, an argument with a data type of DECIMAL(5,2) will result in DECIMAL(6,2). An argument with a data type of DECIMAL(63,2) will result in DECIMAL(63,2).

If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

- Calculate the number 873.726 rounded to 2, 1, 0, -1, -2, -3, and -4 decimal places respectively.

```
SELECT ROUND(873.726,  2),
       ROUND(873.726,  1),
       ROUND(873.726,  0),
       ROUND(873.726, -1),
```

```
       ROUND(873.726, -2),
       ROUND(873.726, -3),
       ROUND(873.726, -4)
  FROM SYSIBM.SYSDUMMY1
```

Returns the following values, respectively:

```
0873.730   0873.700  0874.000  0870.000  0900.000  1000.000  0000.000
```

- Calculate both positive and negative numbers.

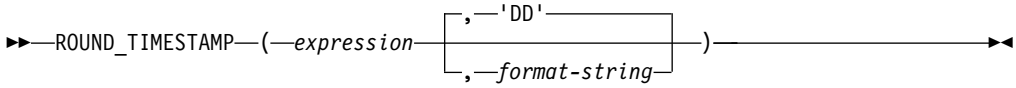
```
SELECT ROUND( 3.5, 0),
       ROUND( 3.1, 0),
       ROUND(-3.1, 0),
       ROUND(-3.5, 0)
  FROM SYSIBM.SYSDUMMY1
```

Returns the following examples, respectively:

```
04.0   03.0  -03.0  -04.0
```

ROUND_TIMESTAMP

The ROUND_TIMESTAMP function returns a timestamp that is the *expression* rounded to the unit specified by the *format-string*. If *format-string* is not specified, *expression* is rounded to the nearest day, as if 'DD' was specified for *format-string*.



expression
An expression that returns a value of one of the following built-in data types: a timestamp, a character-string, or a graphic-string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 83.

format-string
An expression that returns a built-in character string data type or graphic string data type that is not a CLOB or DBCLOB. *format-string* contains a template of how the timestamp represented by *expression* should be rounded. For example, if *format-string* is 'DD', the timestamp that is represented by *expression* is rounded to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid template for a timestamp. The resulting value is then folded to uppercase, so the characters in the value may be in any case. The resulting substring must be a valid format element for a timestamp.

Allowable values for *format-string* are listed in Table 48.

The result of the function is a TIMESTAMP. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Table 48. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
CC SCC	One greater than the first two digits of a four digit year. (Rounds up on the 50th year of the century)	Input value: 1897-12-04-12.22.22.000000 Result: 1901-01-01-00.00.00.000000	Input value: 1897-12-04-12.22.22.000000 Result: 1801-01-01-00.00.00.000000
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (Rounds up on July 1 to January 1st of the next year)	Input value: 1897-12-04-12.22.22.000000 Result: 1898-01-01-00.00.00.000000	Input value: 1897-12-04-12.22.22.000000 Result: 1897-01-01-00.00.00.000000
IYYY IYY IY I	ISO year (Rounds up on July 1 to the first day of the next ISO year. The first day of the ISO year is defined as the Monday of the first ISO week.)	Input value: 1897-12-04-12.22.22.000000 Result: 1898-01-03-00.00.00.000000	Input value: 1897-12-04-12.22.22.000000 Result: 1897-01-04-00.00.00.000000

Table 48. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models (continued)

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
Q	Quarter (Rounds up on the 16th day of the second month of the quarter)	Input value: 1999-06-04-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input value: 1999-06-04-12.12.30.000000 Result: 1999-04-01-00.00.00.000000
MONTH MON MM RM	Month (Rounds up on the 16th day of the month)	Input value: 1999-06-18-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input value: 1999-06-18-12.12.30.000000 Result: 1999-06-01-00.00.00.000000
WW	Same day of the week as the first day of the year (Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the year)	Input value: 2000-05-05-12.12.30.000000 Result: 2000-05-06-00.00.00.000000	Input value: 2000-05-05-12.12.30.000000 Result: 2000-04-29-00.00.00.000000
IW	Same day of the week as the first day of the ISO year (Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the ISO year)	Input value: 2000-05-05-12.12.30.000000 Result: 2000-05-08-00.00.00.000000	Input value: 2000-05-05-12.12.30.000000 Result: 2000-05-01-00.00.00.000000
W	Same day of the week as the first day of the month (Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the month)	Input value: 2000-06-21-12.12.30.000000 Result: 2000-06-22-00.00.00.000000	Input value: 2000-06-22-12.12.30.000000 Result: 2000-06-15-00.00.00.000000
DDD DD J	Day (Rounds up on the 12th hour of the day)	Input value: 2000-05-17-12.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input value: 2000-05-17-12.59.59.000000 Result: 2000-05-17-00.00.00.000000
DAY DY D	Starting day of the week (Rounds up with respect to the 12th hour of the 4th day of the week. The first day of the week is always Sunday)	Input value: 2000-05-17-12.59.59.000000 Result: 2000-05-21-00.00.00.000000	Input value: 2000-05-17-12.59.59.000000 Result: 2000-05-14-00.00.00.000000
HH HH12 HH24	Hour (Rounds up at 30 minutes)	Input value: 2000-05-17-23.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input value: 2000-05-17-23.59.59.000000 Result: 2000-05-17-23.00.00.000000
MI	Minute (Rounds up at 30 seconds)	Input value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.59.00.000000	Input value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.58.00.000000
SS	Second (Rounds up at 500000 microseconds)	Input value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.46.000000	Input value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.45.000000

Table 48. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models (continued)

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
Note: The ISO year starts on the first day of the first ISO week of the year. This can be up to three days before January 1st or three days after January 1st. See "WEEK_ISO" on page 553 for details.			

Example

- Set the host variable RND_TMSTMP with the current year rounded to the nearest month value.

```
SET :RND_TMSTMP = ROUND_TIMESTAMP('2000-03-18-17.30.00', 'MONTH');
```

  Host variable RND_TMSTMP is set with the value 2000-04-01-00.00.00.000000.

ROWID

The ROWID function casts a character string to a row ID.

string-expression
An expression that returns a character string value. Although the string can contain any value, it is recommended that it contain a ROWID value that was previously generated by DB2 for z/OS or DB2 for i to ensure a valid ROWID value is returned. For example, the function can be used to convert a ROWID value that was cast to a CHAR value back to a ROWID value.

If the actual length of *string-expression* is less than 40, the result is not padded. If the actual length of *string-expression* is greater than 40, the result is truncated. If non-blank characters are truncated, a warning is returned.

The length attribute of the result is 40. The actual length of the result is the length of *string-expression*.

The result of the function is a row ID. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

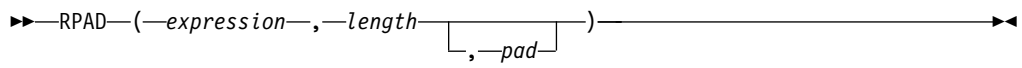
Example

- Assume that table EMPLOYEE contains a ROWID column EMP_ROWID. Also assume that the table contains a row that is identified by a row ID value that is equivalent to X'F0DFD230E3C0D80D81C201AA0A280100000000000203'. Using direct row access, select the employee number for that row.

```
SELECT EMPNO
  FROM EMPLOYEE
  WHERE EMP_ROWID = ROWID(X'F0DFD230E3C0D80D81C201AA0A280100000000000203')
```

RPAD

The RPAD function returns a string composed of *expression* that is padded on the right.



The RPAD function treats leading or trailing blanks in *expression* as significant. Padding will only occur if the actual length of *expression* is less than *length*, and *pad* is not an empty string.

expression
An expression that specifies the string from which the result is derived.

Expression must be a built-in string, numeric, or datetime data type. A numeric or datetime argument is cast to VARCHAR with a CCSID that is the default SBCS CCSID at the current server before evaluating the function. For more information about converting numeric or datetime to a character string, see "VARCHAR" on page 531.

length
An expression that specifies the length of the result. The expression must return a value that is a built-in numeric, character-string, or graphic-string data type. If the data type of the expression is not INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be zero or a positive integer that is less than or equal to *n*, where *n* is the maximum length of the result data type. See Appendix A, "SQL limits," on page 1493 for more information.

If *expression* is a graphic string, *length* indicates the number of DBCS or Unicode graphic characters. If *expression* is a character string, *length* indicates the number of characters where a character may consist of one or more bytes. If *expression* is a binary string, *length* indicates the number of bytes.

pad
An expression that specifies the string with which to pad. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is a numeric or datetime data type, it is implicitly cast to VARCHAR with a CCSID that is the default SBCS CCSID at the current server before evaluating the function.

If *pad* is not specified, the pad character is set as follows:

- For character and graphic strings, a single-byte, double-byte, UTF-16, or UTF-8 blank character based on the data type and CCSID of *expression*.[71]
- For binary strings, hexadecimal zeros.

The value for *expression* and the value for *pad* must have compatible data types. If the CCSID of *pad* is different than the CCSID of *expression*, the *pad* value is converted to the CCSID of *expression*. For more information about data type compatibility, see "Assignments and comparisons" on page 98.

The data type of the result depends on the data type of *expression*:

71. UTF-16 or UCS-2 defines a blank character at code point X'0020' and X'3000'. The database manager pads with the blank at code point X'0020'. The database manager pads UTF-8 with a blank at code point X'20'

Data type of *expression*	Data Type of the Result for RPAD
CHAR or VARCHAR or numeric or datetime	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The length attribute of the result depends on *length*. If *length* is explicitly specified by an integer constant that is greater than zero, the length attribute of the result is *length*. If *length* is explicitly specified by an integer constant that is zero, the length attribute of the result is 1. If *length* is specified as an expression, the length attribute of the result is the minimum of *m*+100 and the maximum length of the result data type, where *m* is the length attribute of *expression*. See Appendix A, "SQL limits," on page 1493 for more information.

The actual length of the result is determined from *length*.

- If *length* is 0, the actual length is 0 and the result is the empty result string.
- If *length* is equal to the actual length of *expression*, the actual length is the length of *expression*.
- If *length* is less than the actual length of *expression*, the result is truncated. The actual length is *length* unless the result data type is varying-length mixed data or varying-length Unicode. In this case, only complete characters will be truncated.
  - For Unicode data, the actual length may be *length*-1 to prevent a double-byte character from being split.
  - For mixed data the actual length may be as little as *length*-3 to account for truncation of a double byte character and possibly a "shift-in" character (X'0F') and a "shift-out" character (X'0E').
- If *length* is greater than the actual length of *expression*, the actual length is *length* unless the result data type is varying-length mixed data or varying-length Unicode and *pad* contains double-byte characters. In this case, only complete characters will be padded.
  - For Unicode data, the actual length may be *length*-1 to prevent a double-byte character from being split.
  - For mixed data, the actual length may be may be as little as *length*-3 to account for truncation of a double byte character and possibly a "shift-in" character (X'0F') and a "shift-out" character (X'0E'). Also, this result will not have redundant shift codes "at the seam". Thus, if the pad is a string ending with a "shift-in" character (X'0F'), and expression begins with a "shift-out" character (X'0E'), these two bytes are eliminated from the result.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of *expression*.

Examples

- *Example 1*: Assume that NAME is a VARCHAR(15) column that contains the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the right with periods:

```
SELECT RPAD(NAME,15,'.' ) AS NAME FROM T1;
```

returns:

```
NAME
---------------
Chris..........
Meg............
Jeff...........
```

- *Example 2*: Assume that NAME is a VARCHAR(15) column that contains the values "Chris", "Meg", and "Jeff". The following query will only pad each value to a length of 5:

```
SELECT RPAD(NAME,5,'.' ) AS NAME FROM T1;
```

returns:

```
NAME
-----
Chris
Meg..
Jeff.
```

- *Example 3*: Assume that NAME is a CHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that the result of RTRIM is a varying length string with the blanks removed

```
SELECT RPAD(RTRIM(NAME),15,'.' ) AS NAME FROM T1;
```

returns:

```
NAME
---------------
Chris..........
Meg............
Jeff...........
```

- *Example 4*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the right with *pad* (note that in some cases, a partial instance of the pad specification is returned):

```
SELECT RPAD(NAME,15,'123' ) AS NAME FROM T1;
```

returns:

```
NAME
---------------
Chris1231231231
Meg123123123123
Jeff12312312312
```

- *Example 5*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that "Chris" is truncated, "Meg" is padded, and "Jeff" is unchanged:

```
SELECT RPAD(NAME,4,'.' ) AS NAME FROM T1;
```

returns:

```
NAME
----
Chri
Meg.
Jeff
```

RRN

The RRN function returns the relative record number of a row.

table-designator
A table designator that could be used to qualify a column in the same relative location in the SQL statement as the RRN function. For more information about table designators, see "Table designators" on page 143.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

The *table-designator* must not identify a *collection-derived-table*.

If the argument identifies a view, common table expression, or nested table expression, the function returns the relative record number of its base table. If the argument identifies a view, common table expression, or nested table expression derived from more than one base table, the function returns the relative record number of the first table in the outer subselect of the view, common table expression, or nested table expression.

If the argument identifies a distributed table, the function returns the relative record number of the row on the node where the row is located. If the argument identifies a partitioned table, the function returns the relative record number of the row in the partition where the row is located. This means that RRN will not be unique for each row of a partitioned or distributed table.

The argument must not identify a view, common table expression, or nested table expression whose outer fullselect subselect includes an aggregate function, a GROUP BY clause, a HAVING clause, a UNION clause, an INTERSECT or EXCEPT clause, DISTINCT clause, or VALUES clause. The RRN function cannot be specified in a SELECT clause if the fullselect contains an aggregate function, a GROUP BY clause, a HAVING clause, or a VALUES clause. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is a decimal with precision 15 and scale 0. The result can be null.

Example

- Return the relative record number and employee name from table EMPLOYEE for those employees in department 20.

```
SELECT RRN(EMPLOYEE), LASTNAME
  FROM EMPLOYEE
  WHERE DEPTNO = 20
```

RTRIM

The RTRIM function removes blanks or hexadecimal zeroes from the end of a string expression.

expression
An expression that returns a value of any built-in numeric or string data type.[72] A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

- If the argument is a binary string, then the trailing hexadecimal zeros (X'00') are removed.
- If the argument is a DBCS graphic string, then the trailing DBCS blanks are removed.
- If the first argument is a Unicode graphic string, then the trailing UTF-16 or UCS-2 blanks are removed.
- If the first argument is a UTF-8 character string, then the trailing UTF-8 blanks are removed.
- Otherwise, trailing SBCS blanks are removed.

The data type of the result depends on the data type of *string-expression*:

Data type of *string-expression*	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The length attribute of the result is the same as the length attribute of *string-expression*. The actual length of the result is the length of the expression minus the number of bytes removed. If all characters are removed, the result is an empty string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The CCSID of the result is the same as that of the string.

Example

- Assume the host variable HELLO of type CHAR(9) has a value of 'Hello '.

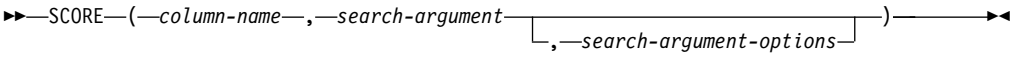
```
SELECT RTRIM(:HELLO)
    FROM SYSIBM.SYSDUMMY1
```

Results in: 'Hello'.

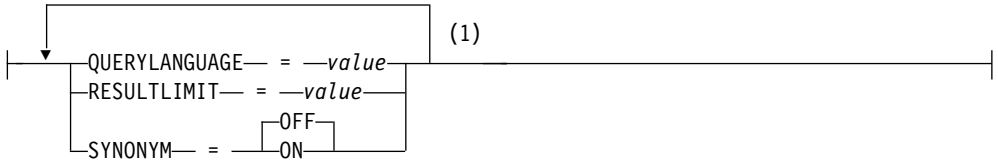
72. The RTRIM function returns the same results as: STRIP(*expression*,TRAILING)

SCORE

The SCORE function searches a text search index using criteria that are specified in a search argument and returns a relevance score that measures how well a document matches the query.



search-argument-options:



Notes:

1 The same clause must not be specified more than once.

column-name
Specifies a qualified or unqualified name of a column that has a text search index that is to be searched. The column must exist in the table or view that is identified in the FROM clause in the statement and the column of the table, or the column of the underlying base table of the view must have an associated text search index. The underlying expression of the column of a view must be a simple column reference to the column of an underlying table, either directly or through another nested view.

search-argument
An expression that returns a character-string data type or graphic-string data type that contains the terms to be searched for. It must not be the empty string or contain all blanks. The actual length of the string must not exceed 32 740 bytes after conversion to Unicode and must not exceed the text search limitations or number of terms as specified in the search argument syntax. For information on *search-argument* syntax, see Appendix G, "Text search argument syntax," on page 1805.

search-argument-options
A character string or graphic string value that specifies the search argument options to use for the search. It must be a constant or a variable.

The options that can be specified as part of the *search-argument-options* are:

QUERYLANGUAGE = *value*
Specifies the language value. The value can be any of the supported language codes. If QUERYLANGUAGE is not specified, the default is the language value of the text search index that is used when the function is invoked. If the language value of the text search index is AUTO, the default value for QUERYLANGUAGE is en_US. For more information on the query language option, see "Text search language options" on page 1814.

RESULTLIMIT = *value*
Specifies the maximum number of results that are to be returned from the

underlying search engine. The *value* must be an integer from 1 to 2 147 483 647. If RESULTLIMIT is not specified, no result limit is in effect for the query.

SCORE may or may not be called for each row of the result table, depending on the plan that the optimizer chooses. If SCORE is called once for the query to the underlying search engine, a result set of all of the ROWIDs or primary keys that match are returned from the search engine. This result set is then joined to the table containing the column to identify the result rows. In this case, the RESULTLIMIT value acts like a FETCH FIRST *n* ROWS ONLY from the underlying text search engine and can be used as an optimization. If SCORE is called for each row of the result because the optimizer determines that is the best plan, then the RESULTLIMIT option has no effect.

SYNONYM = OFF or SYNONYM = ON
Specifies whether to use a synonym dictionary associated with the text search index. The default is OFF.

OFF
Do not use a synonym dictionary.

ON Use the synonym dictionary associated with the text search index.

If *search-argument-options* is the empty string or the null value, the function is evaluated as if *search-argument-options* were not specified.

The result of the function is a double-precision floating-point number. If *search-argument* can be null, the result can be null; if *search-argument* is null, the result is the null value.

The result of SCORE is a value between 0 and 1. The more frequent the column contains a match for the search criteria specified by *search-argument*, the larger the result value. If a match is not found, the result is 0. If the column value is null or *search-argument* contains only blanks or is the empty string, the result is 0.

SCORE is a non-deterministic function.

Notes

Prerequisites: In order to use the CONTAINS and SCORE functions, OmniFind Text Search Server for DB2 for i must be installed and started.

Rules: If a view, nested table expression, or common table expression provides a text search column for a CONTAINS or SCORE scalar function and the applicable view, nested table expression, or common table expression has a DISTINCT clause on the outermost SELECT, the SELECT list must contain all the corresponding key fields of the text search index.

If a view, nested table expression, or common table expression provides a text search column for a CONTAINS or SCORE scalar function, the applicable view, nested table expression, or common table expression cannot have a UNION, EXCEPT, or INTERSECT at the outermost SELECT.

If a common table expression provides a text search column for a CONTAINS or SCORE scalar function, the common table expression cannot be subsequently referenced again in the entire query unless that reference does not provide a text search column for a CONTAINS or SCORE scalar function.

CONTAINS and SCORE scalar functions are not allowed if the query specifies:

- a distributed table,
- a table with a read trigger, or
- a logical file built over multiple physical file members.

Example

- The following statement generates a list of employees in the order of how well their resumes match the query "programmer AND (java OR cobol)", along with a relevance value that is normalized between 0 (zero) and 100.

```
SELECT EMPNO, INTEGER(SCORE(RESUME, 'programmer AND
  (java OR cobol)') * 100) AS RELEVANCE
  FROM EMP_RESUME
  WHERE RESUME_FORMAT = 'ascii'
  AND CONTAINS(RESUME, 'programmer AND (java OR cobol)') = 1
  ORDER BY RELEVANCE DESC
```

  The database manager first evaluates the CONTAINS predicate in the WHERE clause, and therefore, does not evaluate the SCORE function in the SELECT list for every row of the table. In this case, the arguments for SCORE and CONTAINS must be identical.

SECOND

The SECOND function returns the seconds part of a value.

expression
An expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 83.
- If *expression* is a number, it must be a time duration or timestamp duration. For the valid formats of datetime durations, see "Datetime operands and durations" on page 173.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a time, a timestamp, or a valid character-string representation of a time or timestamp:

  The result is the seconds part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration:

  The result is the seconds part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Examples

- Assume that the host variable TIME_DUR (DECIMAL(6,0)) has the value 153045.

```
SELECT SECOND(:TIME_DUR)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 45.
- Assume that the column RECEIVED (TIMESTAMP) has an internal value equivalent to 1988-12-25-17.12.30.000000.

```
SELECT SECOND(RECEIVED)
  FROM IN_TRAY
```

  Returns the value 30.

SIGN

The SIGN function returns an indicator of the sign of expression.

The returned value is:

–1 if the argument is less than zero

-0 if the argument is DECFLOAT negative zero

0 if the argument is zero

1 if the argument is greater than zero

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is converted to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The result has the same data type and length attribute as the argument, except that precision is increased by one if the argument is DECIMAL or NUMERIC and the scale of the argument is equal to its precision. For example, an argument with a data type of DECIMAL(5,5) will result in DECIMAL(6,5). If the precision is already the maximum precision (*mp*), the scale will be decreased by one. For example, DECIMAL(63,63) will result in DECIMAL(63,62).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume that host variable PROFIT is a large integer with a value of 50000.

```
SELECT SIGN(:PROFIT)
  FROM EMPLOYEE
```

  Returns the value 1.

SIN

The SIN function returns the sine of the argument, where the argument is an angle expressed in radians. The SIN and ASIN functions are inverse operations.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is converted to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable SINE is a decimal (2,1) host variable with a value of 1.5.

```
SELECT SIN(:SINE)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 0.99.

SINH

The SINH function returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is converted to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable HSINE is a decimal (2,1) host variable with a value of 1.5.

```
SELECT SINH(:HSINE)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 2.12.

SMALLINT

The SMALLINT function returns a small integer representation.

Numeric to Smallint

String to Smallint

The SMALLINT function returns a small integer representation of

- A number
- A character or graphic string representation of a decimal number
- A character or graphic string representation of an integer
- A character or graphic string representation of a floating-point number
- A character or graphic string representation of a decimal floating-point number

Numeric to Smallint

numeric-expression
An expression that returns a numeric value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error is returned. The fractional part of the argument is truncated.

String to Smallint

string-expression
An expression that returns a value that is a character-string or graphic-string representation of a number.

If the argument is a *string-expression*, the result is the same number that would result from CAST(*string-expression* AS SMALLINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant. If the whole part of the argument is not within the range of small integers, an error is returned. Any fractional part of the argument is truncated.

The result of the function is a small integer. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

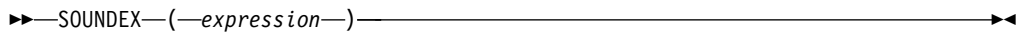
Example

- Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and the employee number (EMPNO).

```
SELECT SMALLINT(SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
  FROM EMPLOYEE
```

SOUNDEX

The SOUNDEX function returns a 4 character code representing the sound of the words in the argument. The result can be used to compare with the sound of other strings.



expression
An expression that returns a value of any built-in numeric or string data type, that is not a CLOB or DBCLOB. The argument cannot be a binary string. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The data type of the result is CHAR(4). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the default CCSID of the current server.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search out words with similar sounds. The comparison can be done directly or by passing the strings as arguments to the DIFFERENCE function. For more information, see "DIFFERENCE" on page 333.

Example

- Using the EMPLOYEE table, find the EMPNO and LASTNAME of the employee with a surname that sounds like 'Loucesy'.

```
SELECT EMPNO, LASTNAME
       FROM EMPLOYEE
       WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy')
```

  Returns the row:

```
000110 LUCCHESSI
```

SPACE

The SPACE function returns a character string that consists of the number of SBCS blanks that the argument specifies.

expression
An expression that returns a value of any built-in SMALLINT, INTEGER, BIGINT, character-string, or graphic-string data type. A string argument is converted to integer before evaluating the function. For more information about converting strings to integer, see "INTEGER or INT" on page 384.

The *expression* specifies the number of SBCS blanks for the result, and it must be between 0 and 32740. If *expression* is a constant, it must not be the constant 0.

The result of the function is a varying-length character string (VARCHAR) that contains SBCS data.

If *expression* is a constant, the length attribute of the result is the constant. Otherwise, the length attribute of the result is 4000. The actual length of the result is the value of *expression*. The actual length of the result must not be greater than the length attribute of the result.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID is the default CCSID for SBCS data of the job.

Example

- The following statement returns a character string that consists of 5 blanks.

```
SELECT SPACE(5)
  FROM SYSIBM.SYSDUMMY1
```

SQRT

The SQRT function returns the square root of a number.

expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is converted to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344. The value of *expression* must be greater than or equal to zero.

If the data type of the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Results involving DECFLOAT special values: For decimal floating-point values the special values are treated as follows:

- SQRT(NaN) returns NaN. [73]
- SQRT(-NaN) returns NaN. [73]
- SQRT(Infinity) returns Infinity.
- SQRT(-Infinity) returns NaN. [73]
- SQRT(sNaN) and SQRT(-sNaN) return a warning or error. [60]

Example

- Assume the host variable SQUARE is a DECIMAL(2,1) host variable with a value of 9.0.

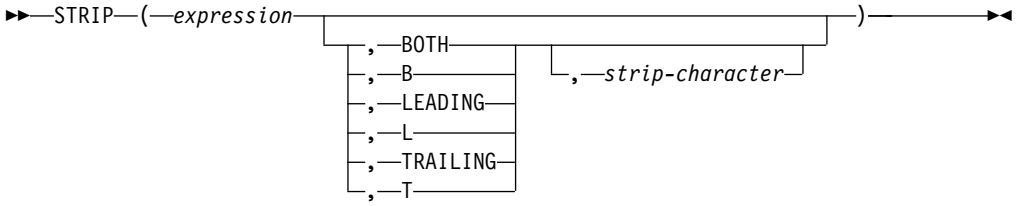
```
SELECT SQRT(:SQUARE)
  FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 3.00.

73. If *YES is specified for the SQL_DECFLOAT_WARNINGS query option, NaN is returned with a warning.

STRIP

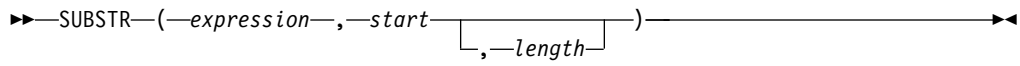
The STRIP function removes blanks or another specified character from the end, the beginning, or both ends of a string expression.



The STRIP function is identical to the TRIM scalar function. For more information, see "TRIM" on page 519.

SUBSTR

The SUBSTR function returns a substring of a string.



expression
An expression that specifies the string from which the result is derived.

Expression must be any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531. If *expression* is a character string, the result of the function is a character string. If it is a graphic string, the result of the function is a graphic string. If it is a binary string, the result of the function is a binary string.

A substring of *expression* is zero or more contiguous characters of *expression*. If *expression* is a graphic string, a character is a DBCS or Unicode graphic character. If *expression* is a character string, a character is a byte.[74] If *expression* is a binary string, a character is a byte.

start
An expression that specifies the position within *expression* of the first character (or byte) of the result. The expression must return a value that is a built-in BIGINT, INTEGER, or SMALLINT data type. A value of 1 indicates that the first character of the result is the first character of *expression*. A negative or zero value indicates a position before the beginning of the string. It may also be greater than the length attribute of *expression*. (The length attribute of a varying-length string is its maximum length.)

length
An expression that specifies the length of the result. If specified, *length* must be an expression that returns a value that is a built-in BIGINT, INTEGER, or SMALLINT data type. The value must be greater than or equal to 0.

If *length* is explicitly specified, *expression* is effectively padded on the right with the necessary number of blank characters so that the specified substring of *expression* always exists. Hexadecimal zeroes are used as the padding character when *expression* is a binary string.

If *expression* is a fixed-length string, omission of *length* is an implicit specification of LENGTH(*expression*) - *start* + 1, which is the number of characters (or bytes) from the *start* character (or byte) to the last character (or byte) of *expression*. If *expression* is a varying-length string, omission of *length* is an implicit specification of zero or LENGTH(*expression*) - *start* + 1, whichever is greater. If the resulting length is zero, the result is the empty string.

The data type of the result depends on the data type of *expression*:

74. The SUBSTR function accepts mixed data strings. However, because SUBSTR operates on a strict byte-count basis, the result will not necessarily be a properly formed mixed data string.

Data type of *expression*	Data Type of the Result for SUBSTR
CHAR or VARCHAR	CHAR, if: • *length* is explicitly specified by an integer constant that is greater than zero. • *length* is not explicitly specified, but *expression* is a fixed-length string and *start* is an integer constant. VARCHAR, in all other cases
CLOB	CLOB
GRAPHIC or VARGRAPHIC	GRAPHIC, if: • *length* is explicitly specified by an integer constant that is greater than zero. • *length* is not explicitly specified, but *expression* is a fixed-length string and *start* is an integer constant. VARGRAPHIC, in all other cases.
DBCLOB	DBCLOB
BINARY or VARBINARY	BINARY, if: • *length* is explicitly specified by an integer constant that is greater than zero. • *length* is not explicitly specified, but *expression* is a fixed-length string and *start* is an integer constant. VARBINARY, in all other cases.
BLOB	BLOB

If *expression* is not a LOB, the length attribute of the result depends on *length*, *start*, and the attributes of *expression*.

- If *length* is explicitly specified by an integer constant that is greater than zero, the length attribute of the result is *length*.
- If *length* is not explicitly specified, but *expression* is a fixed-length string and *start* is an integer constant, the length attribute of the result is LENGTH(*expression*) - *start* + 1.

In all other cases, the length attribute of the result is the same as the length attribute of *expression*. (Remember that if the actual length of *expression* is less than the value for *start*, the actual length of the substring is zero.)

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of *expression*.

Examples

- Assume the host variable NAME (VARCHAR(50)) has a value of 'KATIE AUSTIN' and the host variable SURNAME_POS (INTEGER) has a value of 7.

```
SELECT SUBSTR(:NAME, :SURNAME_POS)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'AUSTIN'.

- Likewise,

```
SELECT SUBSTR(:NAME, :SURNAME_POS, 1)
  FROM SYSIBM.SYSDUMMY1
```

Returns the value 'A'.

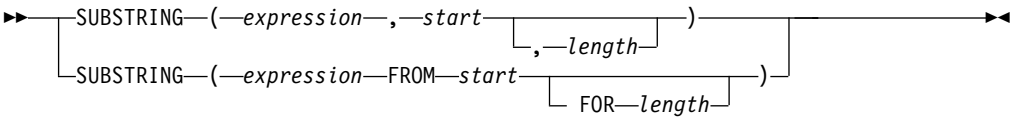
- Select all rows from the PROJECT table for which the project name (PROJNAME) starts with the word 'OPERATION '.

```
SELECT *
  FROM PROJECT
  WHERE SUBSTR(PROJNAME,1,10) = 'OPERATION '
```

The space at the end of the constant is necessary to preclude initial words such as 'OPERATIONS'.

SUBSTRING

The SUBSTRING function returns a substring of a string.



expression
An expression that specifies the string from which the result is derived.

Expression must be any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531. If *expression* is a character string, the result of the function is a character string. If it is a graphic string, the result of the function is a graphic string. If it is a binary string, the result of the function is a binary string.

A substring of *expression* is zero or more contiguous characters of *expression*. If *expression* is a graphic string, a character is a DBCS or Unicode graphic character. If *expression* is a character string, a character is a character that may consist of one or more bytes. If *expression* is a binary string, a character is a byte.

start
An expression that specifies the position within *expression* of the first character (or byte) of the result. The expression must return a value that is a built-in BIGINT, INTEGER, or SMALLINT data type. A value of 1 indicates that the first character of the result is the first character of *expression*. A negative or zero value indicates a position before the beginning of the string. It may also be greater than the length attribute of *expression*. (The length attribute of a varying-length string is its maximum length.)

length
An expression that specifies the maximum actual length of the resulting substring. If specified, *length* must be an expression that returns a value that is a built-in BIGINT, INTEGER, or SMALLINT data type. The value must be greater than or equal to 0.

If *length* is explicitly specified, padding is not performed.

If *expression* is a fixed-length string, omission of *length* is an implicit specification of LENGTH(*expression*) - *start* + 1, which is the number of characters (or bytes) from the *start* character (or byte) to the last character (or byte) of *expression*. If *expression* is a varying-length string, omission of *length* is an implicit specification of zero or LENGTH(*expression*) - *start* + 1, whichever is greater. If the resulting length is zero, the result is the empty string.

The data type of the result depends on the data type of *expression*:

Data type of *expression*	**Data Type of the Result for SUBSTRING**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY

Data type of *expression*	Data Type of the Result for SUBSTRING
BLOB	BLOB

The length attribute of the result is the same as the length attribute of *expression*. (Remember that if the actual length of *expression* is less than the value for *start*, the actual length of the substring is zero.)

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of *expression*.

Examples

- Select all rows from the PROJECT table for which the project name (PROJNAME) starts with the word 'OPERATION '.

```
SELECT *
  FROM PROJECT
  WHERE SUBSTRING(PROJNAME,1,10) = 'OPERATION '
```

  The space at the end of the constant is necessary to preclude initial words such as 'OPERATIONS'.

- Assume that FIRSTNAME is a VARCHAR(12) column, encoded in Unicode UTF-8, in T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

```
SELECT SUBSTRING(FIRSTNAME, 1,2), SUBSTR(FIRSTNAME, 1,2)
  FROM T1
```

  Returns the values 'Jü' (x'4AC3BC') and 'Jô' (x'4AC3').

TAN

The TAN function returns the tangent of the argument, where the argument is an angle expressed in radians. The TAN and ATAN functions are inverse operations.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is converted to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable TANGENT is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT TAN(:TANGENT)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 14.10.

TANH

The TANH function returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians. The TANH and ATANH functions are inverse operations.

expression
An expression that returns a value of any built-in numeric data type (except for DECFLOAT), character-string, or graphic-string data type. A string argument is converted to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable HTANGENT is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT TANH(:HTANGENT)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the approximate value 0.90.

TIME

The TIME function returns a time from a value.

expression
An expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a time. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a time:

  The result is that time.
- If the argument is a timestamp:

  The result is the time part of the timestamp.
- If the argument is a character string:

  The result is the time or time part of the timestamp represented by the character string. When a string representation of a time is SBCS with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a time value.

  When a string representation of a time is mixed data with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a time value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

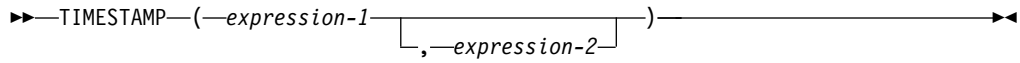
Example

- Select all notes from the IN_TRAY sample table that were received at least one hour later in the day (any day) than the current time.

```
SELECT *
  FROM IN_TRAY
  WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
```

TIMESTAMP

The TIMESTAMP function returns a timestamp from its argument or arguments.



expression-1
If only one argument is specified, the argument must be an expression that returns a value of one of the following built-in data types: a timestamp, a character string, or a graphic string. If *expression-1* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be one of the following:

- A valid string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 83.
- A string with an actual length of 7 that represents a valid date in the form yyyynnn, where yyyy are digits denoting a year, and nnn are digits between 001 and 366 denoting a day of that year.
- A character string with an actual length of 13 that is assumed to be a result from a GENERATE_UNIQUE function. For information on GENERATE_UNIQUE, see "GENERATE_UNIQUE" on page 360.

If both arguments are specified, the first argument must be an expression that returns a value of one of the following built-in data types: a date, a character string, or a graphic string. If *expression-1* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a date.

expression-2
An expression that returns a value of one of the following built-in data types: a time, a character string, or a graphic string.

If *expression-2* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time. For the valid formats of string representations of times, see "String representations of datetime values" on page 83.

The result of the function is a timestamp. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The other rules depend on whether the second argument is specified:

- If both arguments are specified:

  The result is a timestamp with the date specified by the first argument and the time specified by the second argument. The microsecond part of the timestamp is zero.
- If only one argument is specified and it is a timestamp:

  The result is that timestamp.
- If only one argument is specified and it is a character string:

  The result is the timestamp represented by that character string. If the argument is a character string of length 14, the timestamp has a microsecond part of zero.

When a string representation of a date, time, or timestamp is SBCS data with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a timestamp value.

When a string representation of a date, time, or timestamp is mixed data with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a timestamp value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when only one argument is specified. For more information, see "CAST specification" on page 185.

Example

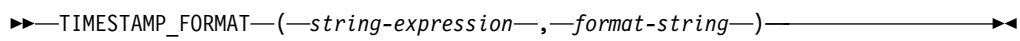
- Assume the following date and time values:

```
SELECT TIMESTAMP( DATE('1988-12-25'), TIME('17.12.30') )
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value '1988-12-25-17.12.30.000000'.

TIMESTAMP_FORMAT

The TIMESTAMP_FORMAT function returns a timestamp that is based on the interpretation of the input string using the specified format.



string-expression
An expression that returns a value of any built-in character string data type or graphic string data type.

The resulting substring is interpreted as a a date or timestamp using the format specified by *format-string*.

format-string
An expression that returns a built-in character string data type or graphic string data type. *format-string* contains a template of how *string-expression* is to be interpreted as a date or a timestamp value.

A valid *format-string* must contain at least one format element, must not contain multiple specifications for any component of a date or a timestamp, and can contain any combination of the format elements, unless otherwise noted in Table 49. For example, *format-string* cannot contain both YY and YYYY, because they are both used to interpret the year component of *string-expression*. Refer to the table to determine which format elements cannot be specified together.

The elements of *format-sting* are not case sensitive. Two format elements can optionally be separated by one or more of the following separator characters:

- minus sign (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semicolon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of *format-string*. These separator characters can be used in any combination in the format string, for example 'YYYY/MM-DD HH:MM.SS'. Separator characters specified in a *string-expression* are used to separate components and are not required to match the separator characters specified in the *format-string*.

Table 49. Format elements for the TIMESTAMP_FORMAT function

Format element	Related components of a timestamp	Description
AM or PM [1, 2]	hour	Meridian indicator (morning or evening) without periods. The meridian indicator is retrieved from message CPX9035 in message file QCPFMSG in library *LIBL.
A.M. or P.M. [1, 2]	hour	Meridian indicator (morning or evening) with periods. This format element uses the exact strings 'A.M.' or 'P.M.' and is independent of the language used for messages in the job.

Table 49. Format elements for the TIMESTAMP_FORMAT function (continued)

Format element	Related components of a timestamp	Description
DAY, Day, or day [1, 3]	none	Name of the day in uppercase, titlecase, or lowercase format. The name of the day is retrieved from message CPX9034 in message file QCPFMSG in library *LIBL.
DY, Dy, or dy [1, 3]	none	Abbreviated name of the day in uppercase, titlecase, or lowercase format. The abbreviated name of the day is retrieved from message CPX9039 in message file QCPFMSG in library *LIBL.
D [1, 3]	none	Day of week (1-7), where 1 is Sunday.
DD	day	Day of month (01-31).
DDD	month, day	Day of year (001-366).
FF or FF*n*	fractional seconds	Fractional seconds (0-999999999999). The number *n* is used to specify the number of digits expected in the *string-expression*. Valid values for *n* are 1-12. Specifying FF is equivalent to specifying FF6. When the component in *string-expression* that corresponds to the FF format element is followed by a separator character or is the last component, the number of digits for the fractional seconds can be less than what is specified by the format element. In this case, zero digits are padded onto the right of the specified digits.
HH	hour	HH behaves the same as HH12.
HH12	hour	Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.
HH24	hour	Hour of the day (00-24) in 24-hour format.
J	year, month, and day	Julian date (number of days since January 1, 4713 BC).
MI	minute	Minute (00-59).
MM	month	Month (01-12).
MONTH, Month, or month [1]	month	Name of the month in uppercase, titlecase, or lowercase format. The name of the month is retrieved from message CPX3BC0 in message file QCPFMSG in library *LIBL.
MON, Mon, or mon [1]	month	Abbreviated name of the month in uppercase, titlecase, or lowercase format. The name of the month is retrieved from message CPX8601 in message file QCPFMSG in library *LIBL.
NNNNNN	microseconds	Microsecond (same as FF6).
RR [4]	year	Last 2 digits of the adjusted year (00-99).
RRRR [4]	year	Four digit adjusted year (0000-9999).
SS	seconds	Seconds (00-59).
SSSSS	hours, minutes, and seconds	Seconds since previous midnight (00000-86400).
Y	year	Last digit of the year (0-9). First three digits of the current year are used to determine the full 4-digit year.
YY	year	Last 2 digits of the year (00-99). First two digits of the current year are used to determine the full 4-digit year.

Table 49. Format elements for the TIMESTAMP_FORMAT function (continued)

Format element	Related components of a timestamp	Description
YYY	year	Last three digits of the year (000-999). First digit of the current year is used to determine the full 4-digit year.
YYYY	year	4-digit year (0000-9999).

Notes:

1. Only these exact spellings and case combinations can be used. If this format element is specified in an invalid case combination an error is returned.
2. The AM and PM set of meridian indicators can be used interchangeably in the *format-string*, as can A.M. and P.M. If HH24 is used in the *format-string* along with a meridian indicator, the value of the meridian indicator in the *string-expression* is not used for determining the hour portion of the resulting timestamp.
3. The DAY, Day, day DY, Dy, dy, and D format elements do not contribute to any components of the resulting timestamp. However, a specified value for any of these format elements must be correct for the combination of the year, month, and day components of the resulting timestamp. For example, a value of 'Monday 2008-10-06' for *string-expression* is valid for a value of 'Day YYYY-MM-DD'. However, a value of 'Tuesday 2008-10-06' for *string-expression* would result in an error for the same *format-string*.
4. The RR and RRRR format elements can be used to alter how a specification for a year is to be interpreted by adjusting the value to produce a 2-digit or a 4-digit value depending on the leftmost two digits of the current year according to the following table:

Last two digits of current year	Two digits of year in *string-expression*	First 2 digits of the year component of date or timestamp
0-50	0-49	First 2 digits of current year
51-99	0-49	First 2 digits of current year + 1
0-50	50-99	First 2 digits of current year - 1
51-99	50-99	First 2 digits of current year

   For example, if the current year is 2007, '86' with format 'RR' means 1986, but if the current year is 2052, it means 2086.

The following defaults will be used when a *format-string* does not include a format element for one of the components of a timestamp:

Timestamp component	Default
year	current year, as 4 digits
month	current month, as 2 digits
day	01 (first day of the month)
hour	00
minute	00
second	00

Timestamp component	Default
microsecond	000000

If *string-expression* does not include a value that corresponds to an hour, minute, second, or microseconds format element that is specified in the *format-string*, these same defaults are used.

Leading zeros can be specified for any component of the date or timestamp value (for example, month, day, hour, minutes, seconds) that does not have the maximum number of significant digits for the corresponding format element in the *format-string*.

A substring of the *string-expression* representing a component of a date or timestamp (such as year, month, day, hour, minutes, seconds) can include less than the maximum number of digits for that component of the date or timestamp. Any missing digits default to zero. For example, with a format-string of 'YYYY-MM-DD HH24:MI:SS', an input value of '999-3-9 5:7:2' would produce the same result as '0999-03-09 05:07:02'.

The result of the function is a timestamp. If the either argument can be null, the result can be null; if the either argument is null, the result is the null value.

Note

Julian and Gregorian calendar: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.

Syntax alternatives: TO_DATE and TO_TIMESTAMP are synonyms for TIMESTAMP_FORMAT.

Example

- Insert a row into the IN_TRAY table with a receiving timestamp that is equal to one second before the beginning of the year 2000 (December 31, 1999 at 23:59:59).

```
INSERT INTO IN_TRAY (RECEIVED)
     VALUES (TIMESTAMP_FORMAT('1999-12-31 23:59:59',
       'YYYY-MM-DD HH24:MI:SS'))
```

- An application receives strings of date information into a variable called INDATEVAR. This value is not strictly formatted and might include two or four digits for years, and one or two digits for months and days. Date components might be separated with minus sign (-) or slash (/) characters and are expected to be in day, month, and year order. Time information consists of hours (in 24-hour format) and minutes, and is usually separated by a colon. Sample values include '15/12/98 13:48' and '9-3-2004 8:02'. Insert such values into the IN_TRAY table.

```
INSERT INTO IN_TRAY (RECEIVED)
     VALUES (TIMESTAMP_FORMAT(:INDATEVAR,
       'DD/MM/RRRR HH24:MI'))
```

  The use of RRRR in the format allows for 2- and 4-digit year values and assigns missing first two digits based on the current year. If YYYY were used, input values with a 2-digit year would have leading zeros. The slash separator also allows the minus sign character. Assuming a current year of 2007, resulting timestamps from the sample values are:

```
'15/12/98 13:48' --> 1998-12-15-13.48.00.000000
'9-3-2004 8:02'  --> 2004-03-09-08.02.00.000000
```

- Set the character variable TVAR to the value of ROUTINE_CREATED from QSYS2.SYSPROCS if it is equal to one second before the beginning of the year 2000 ('1999-12-31 23:59:59'). The character string should be interpreted according to the format string provided.

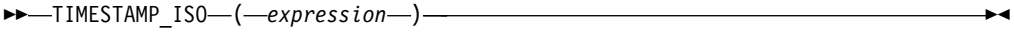
```
SELECT VARCHAR_FORMAT(ROUTINE_CREATED, 'YYYY-MM-DD HH24:MI:SS')
     INTO :TVAR
     FROM QSYS2.SYSPROCS
     WHERE ROUTINE_CREATED =
                TIMESTAMP_FORMAT('1999-12-31 23:59:59', 'YYYY-MM-DD HH24:MI:SS')
```

- Return timestamp values for strings containing meridian indicators:

string-expression	*format-expression*	**Result timestamp value**
'2015-10-28 10:29AM'	'YYYY-MM-DD HH12:MIAM'	2015-10-28-10.29.00.000000
'2015-10-28 10:29PM'	'YYYY-MM-DD HH12:MIAM'	2015-10-28-22.29.00.000000
'2015-10-28 10:29AM'	'YYYY-MM-DD HH24:MIAM'	2015-10-28-10.29.00.000000
'2015-10-28 10:29PM'	'YYYY-MM-DD HH24:MIAM'	2015-10-28-10.29.00.000000
'2015-10-28 22:29AM'	'YYYY-MM-DD HH24:MIAM'	2015-10-28-22.29.00.000000
'2015-10-28 22:29PM'	'YYYY-MM-DD HH24:MIAM'	2015-10-28-22.29.00.000000

TIMESTAMP_ISO

Returns a timestamp value based on a date, time, or timestamp argument. If the argument is a date, it inserts zero for the time and microseconds part of the timestamp. If the argument is a time, it inserts the value of CURRENT DATE for the date part of the timestamp and zero for the microseconds part of the timestamp.



expression
An expression that returns a value of one of the following built-in data types: a timestamp, a date, a time, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a timestamp. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If *expression* is a time, the function is not deterministic.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification" on page 185.

Example

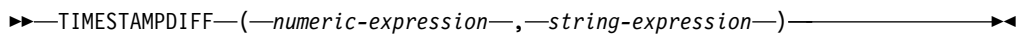
- Assume the following date value:

```
SELECT TIMESTAMP_ISO( DATE( '1988-12-25' ) )
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value '1988-12-25-00.00.00.000000'.

TIMESTAMPDIFF

The TIMESTAMPDIFF function returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.



numeric-expression

The first argument must be a built-in data type of either INTEGER or SMALLINT. The value specifies the interval that is used to determine the difference between two timestamps. Valid values of the interval follow.

Table 50. Valid values for numeric-expression and equivalent intervals that are used to determine the difference between two timestamps

Valid values for *numeric-expression*	**Equivalent intervals**
1	Microseconds
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

string-expression

string-expression is the result of subtracting two timestamps and converting the result to a string of length 22. The argument must be an expression that returns a value of a built-in character string or a graphic string. If *string-expression* is a character or graphic string, it must not be a CLOB or DBCLOB.

If a positive or negative sign is present, it is the first character of the string. The following table describes the elements of the character string duration:

Table 51. TIMESTAMPDIFF String Elements

String elements	**Valid values**	**Character position from the decimal point (negative is left)**
Years	1-9998 or blank	-14 to -11
Months	0-11 or blank	-10 to -9
Days	0-30 or blank	-8 to -7
Hours	0-24 or blank	-6 to -5
Minutes	0-59 or blank	-4 to -3
Seconds	0-59	-2 to -1
Decimal separator	period	0
Microseconds	000000-999999	1 to 6

The result of the function is an integer with the same sign as *string-expression*. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The returned value is determined for each interval as indicated by the following table:

Table 52. TIMESTAMPDIFF Computations

Result interval	Computation using duration elements
Years	years
Quarters	integer value of (months+(years*12))/3
Months	months + (years*12)
Weeks	integer value of ((days+(months*30))/7)+(years*52)
Days	days + (months*30)+(years*365)
Hours	hours + ((days+(months*30)+(years*365))*24)
Minutes (the absolute value of the duration must not exceed 40850913020759.999999)	minutes + (hours+((days+(months*30)+(years*365))*24))*60
Seconds (the absolute value of the duration must be less than 680105031408.000000)	seconds + (minutes+(hours+((days+(months*30)+(years*365))*24))*60)*60
Microseconds (the absolute value of the duration must be less than 3547.483648)	microseconds + (seconds+(minutes*60))*1000000

The following assumptions are used when converting the element values to the requested interval type:

- One year has 365 days.
- One year has 52 weeks.
- One year has 12 months.
- One quarter has 3 months.
- One month has 30 days.
- One week has 7 days.
- One day has 24 hours.
- One hour has 60 minutes.
- One minute has 60 seconds.
- One second has 1000000 microseconds.

The use of these assumptions imply that some result values are an estimate of the interval. Consider the following examples:

- Difference of 1 month where the month has less than 30 days.

  ```
  TIMESTAMPDIFF(16, CHAR(TIMESTAMP('1997-03-01-00.00.00') - TIMESTAMP('1997-02-01-00.00.00') ) )
  ```

  The result of the timestamp arithmetic is a duration of 00000100000000.000000, or 1 month. When the TIMESTAMPDIFF function is invoked with 16 for the interval argument (days), the assumption of 30 days in a month is applied and the result is 30.

- Difference of 1 day less than 1 month where the month has less than 30 days.

```
TIMESTAMPDIFF(16, CHAR(TIMESTAMP('1997-03-01-00.00.00') - TIMESTAMP('1997-02-02-00.00.00') ) )
```

  The result of the timestamp arithmetic is a duration of 00000027000000.000000, or 27 days. When the TIMESTAMPDIFF function is invoked with 16 for the interval argument (days), the result is 27.

- Difference of 1 day less than 1 month where the month has 31 days.

```
TIMESTAMPDIFF(64, CHAR(TIMESTAMP('1997-09-01-00.00.00') - TIMESTAMP('1997-08-02-00.00.00') ) )
```

  The result of the timestamp arithmetic is a duration of 00000030000000.000000, or 30 days. When the TIMESTAMPDIFF function is invoked with 64 for the interval argument (months), the result is 0. The days portion of the duration is 30, but it is ignored because the interval specified months.

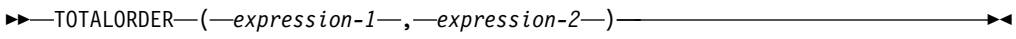
Example

- The following statement estimates the age of employees in months and returns that value as AGE_IN_MONTHS:

```
SELECT
    TIMESTAMPDIFF(64,
    CAST(CURRENT_TIMESTAMP-CAST(BIRTHDATE AS TIMESTAMP) AS CHAR(22)))
      AS AGE_IN_MONTHS
  FROM EMPLOYEE
```

TOTALORDER

The TOTALORDER function returns an ordering for DECFLOAT values.



The TOTALORDER function returns a small integer value that indicates how *expression-1* compares with *expression-2*.

expression-1
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

expression-2
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

Numeric comparison is exact, and the result is determined for finite operands as if range and precision were unlimited. Overflow or underflow cannot occur.

TOTALORDER determines ordering based on the total order predicate rules of IEEE 754R, with the following result:

-1	if the first operand is lower in order compared to the second.
0	if both operands have the same order.
1	if the first operand is higher in order compared to the second.

The ordering of the special values and finite numbers is as follows:

-NAN<-SNAN<-INFINITY<-0.10<-0.100<-0<0<0.100<0.10<INFINITY<SNAN<NAN

The result of the function is SMALLINT. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

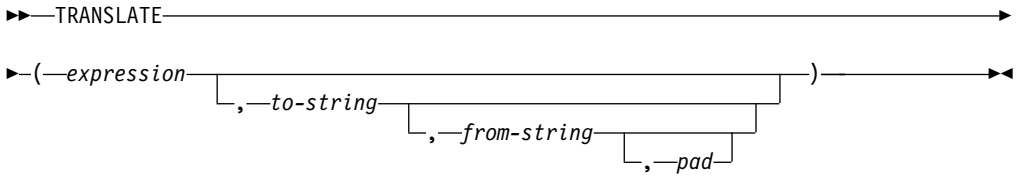
Examples

The following examples show the use of the TOTALORDER function to compare decimal floating-point values:

```
TOTALORDER(-INFINITY, -INFINITY)                =    0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.0))      =    0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.00))     =   -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-0.5))      =   -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(0.5))       =   -1
TOTALORDER(DECFLOAT(-1.0), INFINITY)            =   -1
TOTALORDER(DECFLOAT(-1.0), SNAN)                =   -1
TOTALORDER(DECFLOAT(-1.0), NAN)                 =   -1
TOTALORDER(NAN, DECFLOAT(-1.0))                 =    1
TOTALORDER(-NAN, -NAN)                          =    0
TOTALORDER(-SNAN, -SNAN)                        =    0
TOTALORDER(NAN, NAN)                            =    0
TOTALORDER(SNAN, SNAN)                          =    0
```

TRANSLATE

The TRANSLATE function returns a value in which one or more characters in *expression* may have been converted into other characters.



expression
An expression that specifies the string to be converted *expression* must be any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

to-string
A string that specifies the characters to which certain characters in *expression* are to be converted. This string is sometimes called the *output translation table*. The string must be any built-in numeric or string constant. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531. A character string argument must have an actual length that is not greater than 256.

If the actual length of the *to-string* is less than the actual length of the *from-string*, then the *to-string* is padded to the longer length using either the *pad* character if it is specified or a blank if a *pad* character is not specified. If the actual length of the *to-string* is greater than the actual length of the *from-string*, the extra characters in *to-string* are ignored without warning.

from-string
A string that specifies the characters that if found in *expression* are to be converted. This string is sometimes called the *input translation table*. When a character in *from-string* is found, the character in *expression* is converted to the character in *to-string* that is in the corresponding position of the character in *from-string*

The string must be any built-in numeric or string constant. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531. A character string argument must have an actual length that is not greater than 256.

If there are duplicate characters in *from-string*, the first one scanning from the left is used and no warning is issued. The default value for *from-string* is a string starting with the character X'00' and ending with the character X'FF' (decimal 255).

pad
A string that specifies the character with which to pad *to-string* if its length is less than *from-string*. The string must be a character string constant with a length of 1. The default is an SBCS blank.

If the first argument is a Unicode graphic or UTF-8 string, no other arguments may be specified.

If only the first argument is specified, the SBCS characters of the argument are converted to uppercase, based on the CCSID of the argument. Only SBCS characters are converted. The characters a-z are converted to A-Z, and characters with diacritical marks are converted to their uppercase equivalent, if any. If the first argument is UTF–16, UCS-2, or UTF-8, the alphabetic UTF-16, UCS-2, or UTF-8 characters are converted to uppercase. Refer to the UCS-2 level 1 mapping tables topic of the Globalization topic collection for a description of the monocasing tables that are used for this conversion.

If more than one argument is specified, the result string is built character by character from *expression*, converting characters in *from-string* to the corresponding character in *to-string*. For each character in *expression*, the same character is searched for in *from-string*. If the character is found to be the *n*th character in *from-string*, the resulting string will contain the *n*th character from *to-string*. If *to-string* is less than *n* characters long, the resulting string will contain the pad character. If the character is not found in *from-string*, it is moved to the result string unconverted.

Conversion is done on a byte basis and, if used improperly, may result in an invalid mixed string. The SRTSEQ attribute does not apply to the TRANSLATE function.

The result of the function has the same data type, length attribute, actual length, and CCSID as the argument. If the first argument can be null, the result can be null. If the argument is null, the result is the null value.

Examples

- Monocase the string 'abcdef'.

```
SELECT TRANSLATE('abcdef')
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'ABCDEF'.

- Monocase the mixed character string.

```
SELECT TRANSLATE('ab<SO>c<SI>def')
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'AB<SO>C<SI>DEF'

- Given that the host variable SITE is a varying-length character string with a value of 'Pivabiska Lake Place'.

```
SELECT TRANSLATE(:SITE, '$', 'L')
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'Pivabiska $ake Place'.

```
SELECT TRANSLATE(:SITE, '$$', 'Ll')
  FROM SYSIBM.SYSDUMMY1
```

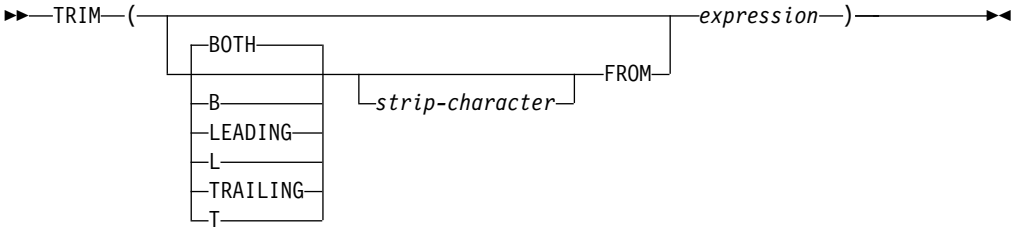
  Returns the value 'Pivabiska $ake P$ace'.

```
SELECT TRANSLATE(:SITE, 'pLA', 'Place', '.')
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'pivAbiskA LAk. pLA..'.

TRIM

The TRIM function removes blanks or another specified character from the end, from the beginning, or from both of a string expression.



The first argument, if specified, indicates whether characters are removed from the end or beginning of the string. If the first argument is not specified, then the characters are removed from both the end and the beginning of the string.

strip-character
The second argument, if specified, is a single-character constant that indicates the binary, SBCS, or DBCS character that is to be removed. If *expression* is a binary string, the second argument must be a binary string constant. If *expression* is a DBCS graphic or DBCS-only string, the second argument must be a graphic constant consisting of a single DBCS character. If the second argument is not specified then:

- If *expression* is a binary string, then the default strip character is a hexadecimal zero (X'00').
- If *expression* is a DBCS graphic string, then the default strip character is a DBCS blank.
- If *expression* is a Unicode graphic string, then the default strip character is a UTF-16 or UCS-2 blank.
- If *expression* is a UTF-8 character string, then the default strip character is a UTF-8 blank.
- Otherwise, the default strip character is an SBCS blank.

expression
An expression that returns a value of any built-in numeric or string data type. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The data type of the result depends on the data type of *expression*:

Data type of *expression*	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The length attribute of the result is the same as the length attribute of *expression*. The actual length of the result is the length of the expression minus the number of bytes removed. If all characters are removed, the result is an empty string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The CCSID of the result is the same as that of the string.

The SRTSEQ attribute does not apply to the TRIM function.

Examples

- Assume the host variable HELLO of type CHAR(9) has a value of ' Hello '.

```
SELECT TRIM(:HELLO), TRIM( TRAILING FROM :HELLO)
  FROM SYSIBM.SYSDUMMY1
```

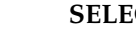
  Results in 'Hello' and ' Hello' respectively.

- Assume the host variable BALANCE of type CHAR(9) has a value of '000345.50'.

```
SELECT TRIM( L '0' FROM :BALANCE )
  FROM SYSIBM.SYSDUMMY1
```

  Results in: '345.50'

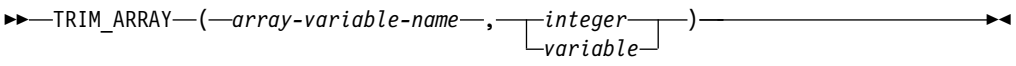
- Assume the string to be stripped contains mixed data.

  SELECT TRIM(BOTH ' **FROM** '  **)**
  FROM SYSIBM.SYSDUMMY1

  Results in: ''

TRIM_ARRAY

The TRIM_ARRAY function returns a copy of the array argument from which the specified number of elements have been removed from the end of the array.



array-variable-name
Identifies an SQL variable or parameter. The variable or parameter must be of an array type.

integer **or** *variable*
An integer or SQL variable or parameter that is a built-in integer data type that specifies the number of elements that will be trimmed from the copy of *array-variable-name*. The cardinality of the result is decreased by the number of elements trimmed.

The result array type is identical to the array type of the first argument.

The result can be null; if either argument is null, the result is the null value.

TRIM_ARRAY can only be used in an SQL procedure as the only expression on the right side of an *assignment-statement*.

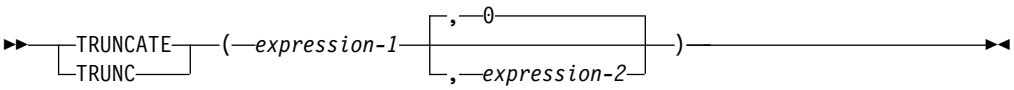
Example

Remove the last element from SQL array variable *p_phonenumbers*.

```
SET p_phonenumbers = TRIM_ARRAY(p_phonenumbers, 1)
```

TRUNCATE or TRUNC

The TRUNCATE function returns *expression–1* truncated to some number of places to the right or left of the decimal point.



expression–1
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A string argument is converted to double-precision floating point before evaluating the function. For more information about converting strings to double-precision floating point, see "DOUBLE_PRECISION or DOUBLE" on page 344.

If *expression-1* is a decimal floating-point data type, the DECFLOAT ROUNDING MODE will not be used. The rounding behavior of TRUNCATE corresponds to a value of ROUND_DOWN. If a different rounding behavior is wanted, use the QUANTIZE function.

expression–2
An expression that returns a value of a built-in small integer, large integer, or big integer data type. The absolute value of integer specifies the number of places to the right of the decimal point for the result if *expression–2* is not negative, or to the left of the decimal point if *expression–2* is negative.

If *expression–2* is not negative, *expression–1* is truncated to the *expression–2* number of places to the right of the decimal point.

If *expression–2* is negative, *expression–1* is truncated to the absolute value of *expression–2*+1 number of places to the left of the decimal point.

If *expression–2* is not specified, *expression–1* is truncated to zero places to the left of the decimal point.

If the absolute value of *expression–2* is larger than the number of digits to the left of the decimal point, the result is 0. For example, TRUNCATE(748.58,-4) = 0.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument.

If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

- Calculate the average monthly salary for the highest paid employee. Truncate the result to two places to the right of the decimal point.

```
SELECT TRUNCATE(MAX(SALARY/12) , 2)
  FROM EMPLOYEE
```

  Because the highest paid employee in the sample employee table earns $52750.00 per year, the example returns the value 4395.83.

- Calculate the number 873.726 truncated to 2, 1, 0, -1, -2, and -3 decimal places respectively.

```
SELECT TRUNCATE(873.726,  2),
       TRUNCATE(873.726,  1),
       TRUNCATE(873.726,  0),
```

```
       TRUNCATE(873.726, -1),
       TRUNCATE(873.726, -2),
       TRUNCATE(873.726, -3)
  FROM SYSIBM.SYSDUMMY1
```

Returns the following values respectively:

```
0873.720   0873.700  0873.000  0870.000  0800.000  0000.000
```

- Calculate both positive and negative numbers.

```
SELECT TRUNCATE( 3.5, 0),
       TRUNCATE( 3.1, 0),
       TRUNCATE(-3.1, 0),
       TRUNCATE(-3.5, 0)
  FROM SYSIBM.SYSDUMMY1
```

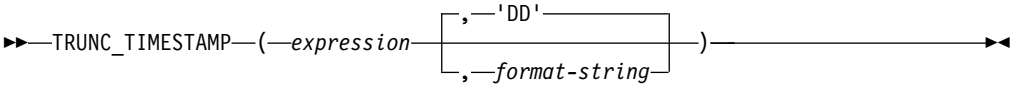
This example returns:

```
3.0   3.0  -3.0  -3.0
```

respectively.

TRUNC_TIMESTAMP

The TRUNC_TIMESTAMP function returns a timestamp that is the *expression* truncated to the unit specified by the *format-string*. If *format-string* is not specified, *expression* is truncated to the nearest day, as if 'DD' was specified for *format-string*.



expression
An expression that returns a value of one of the following built-in data types: a timestamp, a character-string, or a graphic-string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 83.

format-string
An expression that returns a built-in character string data type or graphic string data type that is not a CLOB or DBCLOB. *format-string* contains a template of how the timestamp represented by *expression* should be truncated. For example, if *format-string* is 'DD', the timestamp that is represented by *expression* is truncated to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid template for a timestamp. The resulting value is then folded to uppercase, so the characters in the value may be in any case. The resulting substring must be a valid format element for a timestamp.

Allowable values for *format-string* are listed in Table 48 on page 475.

The result of the function is a TIMESTAMP. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Example

- Set the host variable TRN_TMSTMP with the current year rounded to the nearest year value.

```
SET :TRN_TMSTMP = TRUNC_TIMESTAMP('2000-03-14-17.30.00', 'YEAR');
```

Host variable TRN_TMSTMP is set with the value 2000-01-01-00.00.00.000000.

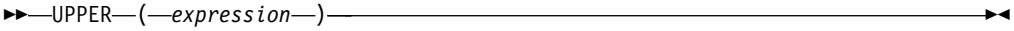
UCASE

The UCASE function returns a string in which all the characters have been converted to uppercase characters, based on the CCSID of the argument.

The UCASE function is identical to the UPPER function. For more information, see "UPPER" on page 526.

UPPER

The UPPER function returns a string in which all the characters have been converted to uppercase characters, based on the CCSID of the argument. Only SBCS and Unicode graphic characters are converted. The characters a-z are converted to A-Z, and characters with diacritical marks are converted to their uppercase equivalent, if any.



Refer to the UCS-2 level 1 mapping tables topic of the Globalization topic collection for a description of the monocasing tables that are used for this translation.

expression
An expression that specifies the string to be converted. *expression* must be any built-in numeric, character, Unicode graphic string. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The result of the function has the same data type, length attribute, actual length, and CCSID as the argument. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: UCASE is a synonym for UPPER.

Examples

- Uppercase the string 'abcdef' using the UPPER scalar function.

```
SELECT UPPER('abcdef')
    FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'ABCDEF'.

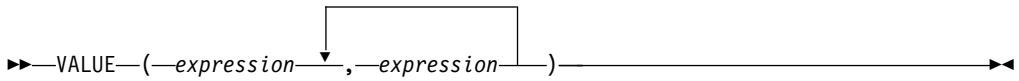
- Uppercase the mixed character string using the UPPER scalar function.

```
SELECT UPPER( 'ab<SO>C<SI>def' )
    FROM SYSIBM.SYSDUMMY1
```

  Returns the value: 'AB<SO>C<SI>DEF'

VALUE

The VALUE function returns the value of the first non-null expression.



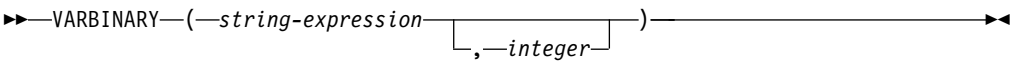
The VALUE function is identical to the COALESCE scalar function. For more information, see "COALESCE" on page 292.

Note

Syntax alternatives: COALESCE should be used for conformance to the SQL 2003 standard.

VARBINARY

The VARBINARY function returns a VARBINARY representation of a string of any type.



The result of the function is VARBINARY. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

string-expression
A *string-expression* whose value can be a character string, graphic string, binary string, or row ID.

integer
An integer constant that specifies the length attribute for the resulting binary string. The value must be between 1 and 32740 (32739 if nullable).

If *integer* is not specified:

- If the *string-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument, unless the argument is a graphic string. In this case, the length attribute of the result is twice the length attribute of the argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of the expression (or twice the length of the expression when the input is graphic data). If the length of the *string-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the first input argument is a character string and all the truncated characters are blanks, or the first input argument is a graphic string and all the truncated characters are double-byte blanks, or the first input argument is a binary string and all the truncated bytes are hexadecimal zeroes.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the length is specified. For more information, see "CAST specification" on page 185.

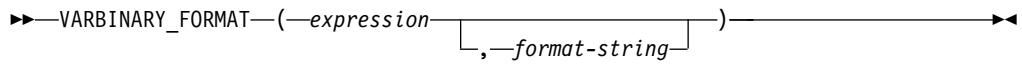
Example

- The following function returns a VARBINARY for the string 'This is a VARBINARY'.

```
SELECT VARBINARY('This is a VARBINARY')
  FROM SYSIBM.SYSDUMMY1
```

VARBINARY_FORMAT

The VARBINARY_FORMAT function returns a binary string representation of a character string that has been formatted using a *format-string*.



expression
An expression that returns a value of any built-in numeric, character-string, or graphic-string data type. A numeric or graphic argument is cast to a character string before evaluating the function. For more information about converting numeric or graphic to a character string, see "VARCHAR" on page 531.

All leading and trailing blanks are removed from *expression* before evaluating the function.

If a *format-string* is specified, the length of *expression* must be equal to the length of the *format-string* and the value of *expression* must conform to the template specified by the *format-string*. If a *format-string* is not specified, the value of *expression* (after removing leading and trailing blanks) should be an even number of characters from the ranges '0' to '9', 'a' to 'f', and 'A' to 'F'. If the length is an odd number of characters, the string is padded on the right with one '0' character.

format-string
An expression that returns a built-in character string or graphic string data type. *format-string* contains a template for how the value for *expression* is to be interpreted.

The valid format strings are: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' and 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' where each 'x' or 'X' corresponds to one hexadecimal digit in the result. If 'X' is specified, the corresponding hexadecimal digit must not be a lower case character. If 'x' is specified, the corresponding hexadecimal digit must not be an upper case character.

The result of the function is a varying-length binary string. The length attribute of the result is half the length attribute of *expression*. If a *format-string* is not specified, the actual length is half the actual length of *expression* (after leading and trailing blanks have been removed and padding to an even number of characters). If a *format-string* is specified, the actual length is half the actual length of the *format-string* (after removing the non-digit separator characters). If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Note

Syntax alternatives: HEXTORAW is a synonym for VARBINARY_FORMAT except that if the length of *expression* is an odd number of characters, the string is padded on the left with one '0' character. VARCHAR_BIT_FORMAT is a synonym for VARBINARY_FORMAT except that the result of the function is a varying-length character string FOR BIT DATA.

Example

- Represent a Universal Unique Identifier in its binary form:

```
VALUES VARBINARY_FORMAT('d83d6360-1818-11db-9804-b622a1ef5492',
                        'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
```

Result returned:

```
BX'D83D6360181811DB9804B622A1EF5492'
```

- Represent a Universal Unique Identifier in its binary form:

```
VALUES VARBINARY_FORMAT('D83D6360-1818-11DB-9804-B622A1EF5492',
                        'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

Result returned:

```
BX'D83D6360181811DB9804B622A1EF5492'
```

- Represent a string of hexadecimal characters in binary form:

```
VALUES VARBINARY_FORMAT('ef01abC9')
```

Result returned:

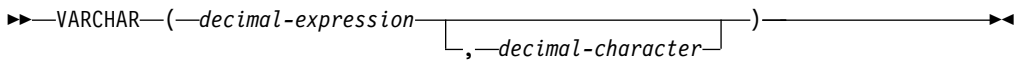
```
BX'EF01ABC9'
```

VARCHAR

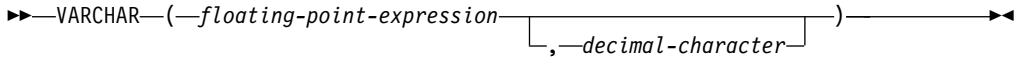
The VARCHAR function returns a character-string representation.

Integer to Varchar

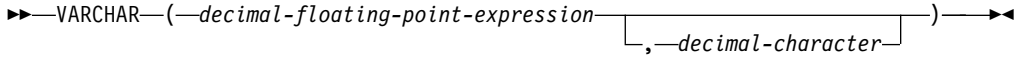
Decimal to Varchar



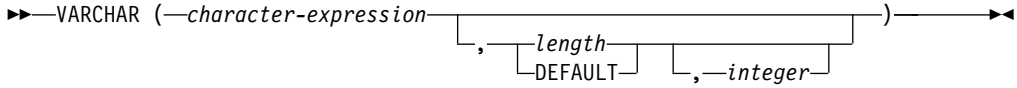
Floating-point to Varchar



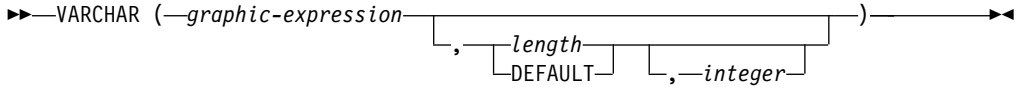
Decimal floating-point to Varchar



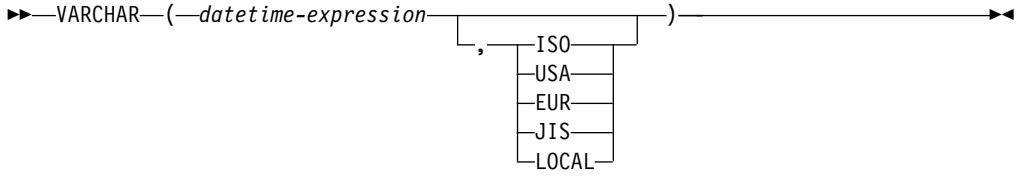
Character to Varchar



Graphic to Varchar



Datetime to Varchar



The VARCHAR function returns a character-string representation of:

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT.
- A decimal number if the first argument is a packed or zoned decimal number.
- A double-precision floating-point number if the first argument is a DOUBLE or REAL.

- A decimal floating-point number if the first argument is DECFLOAT.
- A character string if the first argument is any type of character string.
- A graphic string if the first argument is any graphic string.
- A date value if the first argument is a DATE.
- A time value if the first argument is a TIME.
- A timestamp value if the first argument is a TIMESTAMP.

The result of the function is a varying-length string. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Integer to Varchar

integer-expression
An expression that returns a value that is an integer data type (either SMALLINT, INTEGER, or BIGINT).

The result is a varying-length character string of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. It is left justified.

- If the argument is a small integer, the length attribute of the result is 6.
- If the argument is a large integer, the length attribute of the result is 11.
- If the argument is a big integer, the length attribute of the result is 20.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit or the *decimal-character*.

The CCSID of the result is the default SBCS CCSID at the current server.

Decimal to Varchar

decimal-expression
An expression that returns a value that is a packed or zoned decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is wanted, the DECIMAL scalar function can be used to make the change.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length character string representation of the argument. The result includes a decimal character and up to *p* digits, where *p* is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length attribute of the result is 2+*p* where *p* is the precision of the *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing characters are included. Leading zeros are not included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit or the *decimal-character*.

The CCSID of the result is the default SBCS CCSID at the current server.

Floating-point to Varchar

floating-point expression
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length character string representation of the argument in the form of a floating-point constant.

The length attribute of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0E0.

The CCSID of the result is the default SBCS CCSID at the current server.

Decimal floating-point to Varchar

decimal-floating-point expression
An expression that returns a value that is a decimal floating-point data type.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length character string representation of the argument in the form of a decimal floating-point constant.

The length attribute of the result is 42. The actual length of the result is the smallest number of characters that represents the value of the argument, including the sign, digits, and *decimal-character*. Trailing zeros are significant. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0.

If the DECFLOAT value is Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, a minus sign will be the first character in the string. The DECFLOAT special value sNaN does not result in an exception when converted to a string.

The CCSID of the result is the default SBCS CCSID at the current server.

Character to Varchar

character-expression
An expression that returns a value that is a built-in CHAR, VARCHAR, or CLOB data type.[75]

length
An integer constant that specifies the length attribute for the resulting varying length character string. The value must be between 1 and 32740 (32739 if nullable). If the first argument is mixed data, the second argument cannot be less than 4.

If the second argument is not specified or DEFAULT is specified:

- If the *character-expression* is an empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID of the result. It must be a valid SBCS CCSID, mixed data CCSID, or 65535 (bit data). If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. If the third argument is 65535, then the result is bit data. If the third argument is a SBCS CCSID, then the first argument cannot be a DBCS-either or DBCS-only string.

If the third argument is not specified then:

- If the first argument is SBCS data, then the result is SBCS data. The CCSID of the result is the same as the CCSID of the first argument.
- If the first argument is mixed data (DBCS-open, DBCS-only, or DBCS-either), then the result is mixed data. The CCSID of the result is the same as the CCSID of the first argument.

Graphic to Varchar

graphic-expression
An expression that returns a value that is a GRAPHIC, VARGRAPHIC, and DBCLOB data type. It must not be DBCS-graphic data.

length
An integer constant that specifies the length attribute for the resulting varying length character string. The value must be between 1 and 32740 (32739 if nullable). If the first argument contains DBCS data, the second argument cannot be less than 4.

If the second argument is not specified or DEFAULT is specified, the length attribute of the result is determined as follows (where *n* is the length attribute of the first argument):

- If the *graphic-expression* is the empty graphic string constant, the length attribute of the result is 1.

75. A binary string is also allowed if a CCSID is not specified or if a CCSID of 65535 is explicitly specified.

- If the result is SBCS data, the result length is *n*.
- If the result is mixed data, the result length is (2.5*(*n*-1)) + 4.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID of the result. It must be a valid SBCS CCSID or mixed data CCSID. If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. The third argument cannot be 65535.

If the third argument is not specified, the CCSID of the result is the default CCSID at the current server. If the default CCSID is mixed data, then the result is mixed data. If the default CCSID is SBCS data, then the result is SBCS data.

Datetime to Character

datetime-expression
An expression that is one of the following three built-in data types

date The result is the character-string representation of the date in the format specified by the second argument. If the second argument is not specified, the format used is the default date format. If the format is ISO, USA, EUR, or JIS, the length attribute and actual length of the result is 10. Otherwise the length attribute and actual length of the result is the length of the default date format. For more information see "String representations of datetime values" on page 83.

time The result is the character-string representation of the time in the format specified by the second argument. If the second argument is not specified, the format used is the default time format. The length attribute and actual length of the result is 8. For more information see "String representations of datetime values" on page 83.

timestamp
The second argument is not applicable and must not be specified.

The result is the character-string representation of the timestamp. The length attribute and actual length of the result is 26.

The CCSID of the string is the default SBCS CCSID at the current server.

ISO, EUR, USA, or JIS
Specifies the date or time format of the resulting character string. For more information, see "String representations of datetime values" on page 83.

LOCAL
Specifies that the date or time format of the resulting character string should come from the DATFMT, DATSEP, TIMFMT, and TIMSEP attributes of the job at the current server.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the first argument is a string and the length argument is specified. For more information, see "CAST specification" on page 185.

Example

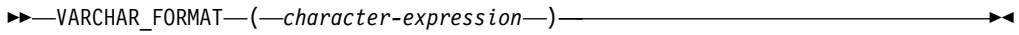
- Make EMPNO varying-length with a length of 10.

```
SELECT VARCHAR(EMPNO,10)
  INTO :VARHV
  FROM EMPLOYEE
```

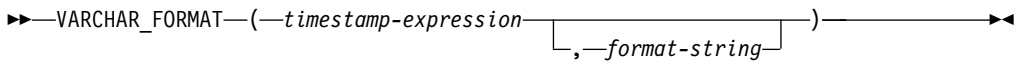
VARCHAR_FORMAT

The VARCHAR_FORMAT function returns a character representation of the first argument in the format indicated by the optional *format-string*.

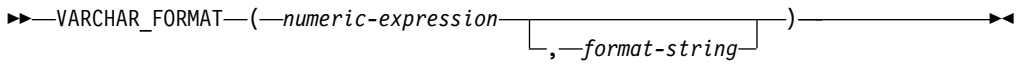
Character to Varchar



Timestamp to Varchar



Numeric to Varchar



If any argument of the VARCHAR_FORMAT function can be null, the result can be null; if any argument is null, the result is the null value.

Character to Varchar

character-expression
An expression that returns a value that is a built-in character-string or graphic-string data type.

If the argument is a character string:

- The length attribute of the result and the actual length are determined as follows:
  - If *character-expression* is an empty string constant, the length attribute of the result is 1.
  - Otherwise, the length attribute and the actual length of the result is the same as the length attribute of the argument.
  - The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*.
- The CCSID of the result is determined as follows:
  - If the argument is SBCS data, then the result is SBCS data. The CCSID of the result is the same as the CCSID of the argument.
  - If the argument is mixed data (DBCS-open, DBCS-only, or DBCS-either), then the result is mixed data. The CCSID of the result is the same as the CCSID of the argument.

If the argument is a graphic string, it must not be DBCS-graphic data.

- The length attribute and the actual length of the result is determined as follows (where *n* is the length attribute of *character-expression*):
  - If *character-expression* is the empty graphic string constant, the length attribute of the result is 1.
  - If the result is SBCS data, the length attribute of the result is *n*.
  - If the result is mixed data, the length attribute of the result is (2.5*(*n*-1)) + 4.

  - The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*.
- The CCSID of the result is the default CCSID at the current server. If the default CCSID is mixed data, then the result is mixed data. If the default CCSID is SBCS data, then the result is SBCS data.

Timestamp to Varchar

timestamp-expression
An expression that returns a value of one of the following built-in data types: a timestamp, a character string, or a graphic string.

If *timestamp-expression* is a character or graphic string, the value of *expression* must be a valid string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 83.

If the argument is a string, the *format-string* argument must also be specified.

format-string
An expression that returns a built-in character string data type or graphic string data type. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. *format-string* contains a template of how *timestamp-expression* is to be formatted. The resulting value may contain characters in any case. A valid format is any combination of the formats listed below optionally separated by valid separators. Valid separators are:

- minus sign (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semicolon (;)
- colon (:)
- blank ( )

Table 53. Format elements for the VARCHAR_FORMAT (Timestamp to VARCHAR) function

Format	Unit
AM or PM [1, 2]	Meridian indicator (morning or evening) without periods. The meridian indicator that is returned is based on the language used for messages in the job. This meridian indicator is retrieved from message CPX9035 in message file QCPFMSG in library *LIBL.
A.M. or P.M. [1, 2]	Meridian indicator (morning or evening) with periods. This format element uses the exact strings 'A.M.' or 'P.M.' and is independent of the language used for messages in the job.
CC	Century (00-99). If the last two digits of the four digit year are zero, the result is the first two digits of the year. Otherwise, the result is the first two digits of the year plus one.
DAY, Day, or day [1, 2]	Name of the day in uppercase, titlecase, or lowercase format. The name of the day that is returned is based on the language used for messages in the job. This name of the day is retrieved from message CPX9034 in message file QCPFMSG in library *LIBL.
DY, Dy, or dy [1, 3]	Abbreviated name of the day in uppercase, titlecase, or lowercase format. The abbreviated name of the day is retrieved from message CPX9039 in message file QCPFMSG in library *LIBL.

Table 53. Format elements for the VARCHAR_FORMAT (Timestamp to VARCHAR) function (continued)

Format	Unit
D [1]	Day of week (1-7), where 1 is Sunday.
DD	Day of month (01-31).
DDD	Day of year (001-366).
FF or FF*n*	Fractional seconds (0-999999). The number *n* is used to specify the number of digits to include in the value returned. Valid values for *n* are 1-6. The default is 6.
HH	HH behaves the same as HH12.
HH12	Hour of the day (01-12) in 12-hour format.
HH24	Hour of the day (00-24) in 24-hour format.
ID	ISO day of week (1-7), where 1 is Monday and 7 is Sunday
IW	ISO week of year (01-53). The week starts on Monday and includes 7 days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week of the year to contain January 4
I	ISO year (0-9). The last digit of the year based on the ISO week that is returned.
IY	ISO year (00-99). The last two digits of the year based on the ISO week that is returned.
IYY	ISO year (000-999). The last three digits of the year based on the ISO week that is returned.
IYYY	ISO year (0000-9999). The year based on the ISO week that is returned.
J	Julian date (0000000-9999999).
MI	Minute (00-59).
MM	Month (01-12).
MONTH, Month, or month [1, 3]	Name of the month in uppercase, titlecase, or lowercase format. The name of the month that is returned is based on the language used for messages in the job. This name of the month is retrieved from message CPX3BC0 in message file QCPFMSG in library *LIBL.
MON, Mon, or mon [1, 3]	Abbreviated name of the month in uppercase, titlecase, or lowercase format. The name of the month that is returned is based on the language used for messages in the job. This name of the month is retrieved from message CPX8601 in message file QCPFMSG in library *LIBL.
NNNNNN	Microseconds (000000-999999). Same as FF6.
Q	Quarter (1-4).
RR	RR behaves the same as YY.
RRRR	RRRR behaves the same as YYYY.
SS	Seconds (00-59).
SSSSS	Seconds since previous midnight (00000-86400).
W	Week of month (1-5). Week 1 starts on the first day of the month and ends on the seventh day.
WW	Week of the year (01-53), where week 1 starts on January 1 and ends on January 7.
Y	Last digit of the year (0-9).

Table 53. Format elements for the VARCHAR_FORMAT (Timestamp to VARCHAR) function (continued)

Format	Unit
YY	Last two digits of the year (00-99).
YYY	Last three digits of the year (000-999).
YYYY	Year (0000-9999).

Notes:

1. This format element is case sensitive. In cases where the format elements are ambiguous, the case insensitive format elements will be considered first.
2. The AM and PM set of meridian indicators can be used interchangeably in the *format-string*, as can A.M. and P.M. The result string will contain the appropriate meridian indicator for the actual time value.
3. Only these exact spellings and case combinations can be used. If this format element is specified in an invalid case combination an error is returned.

Examples of valid format strings are:

```
'HH24-MI-SS'
'HH24-MI-SS-NNNNNN'
'YYYY-MM-DD'
'YYYY-MM-DD-HH24-MI-SS'
'YYYY-MM-DD-HH24-MI-SS-NNNNNN'
'FF3.J/Q-YYYY'
```

The result is a representation of *timestamp-expression* in the format specified by *format-string*. *format-string* is interpreted as a series of format elements that can be separated by one or more separator characters. A string of characters in *format-string* is interpreted as the longest format element that matches an element in the previous table. If two format elements are composed of the same character and they are not separated by a separator character, DB2 interprets the specification starting from the left, as the longest element that matches an element from the previous table, and continues until matches are found for the remainder of the format string.For example, DDYYYY would be interpreted as DD followed by YYYY, rather than D followed by DY, followed by YYY.

The data type of the result is same as the data type of the *format-string*. The length attribute of the result is the maximum of 255 and the length attribute of the *format-string*. *format-string* also determines the actual length of the result. The actual length must not be greater than the length attribute of the result.

The CCSID of the result is same as the CCSID of the *format-string*. If *format-string* is not specified, the CCSID of the result is the default SBCS CCSID at the current server.

Numeric to Varchar

numeric-expression
An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

format-string
An expression that returns a built-in character string, graphic string, or

numeric data type. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. *format-string* contains a template of how *numeric-expression* is to be formatted. A *format-string* must contain a valid combination of the listed format elements according to the following rules:

- A sign format element ('S', 'MI', 'PR') can be specified only one time.
- A decimal point format element can be specified only one time.
- Alphabetic format elements must be specified in upper case.
- A prefix format element can only be specified at the beginning of the format string, before any format elements that are not prefix format elements. When multiple prefix format elements are specified they can be specified in any order.
- A suffix format element can only be specified at the end of the format string, after any format elements that are not suffix format elements. When multiple suffix format elements are specified they can be specified in any order.
- A comma or G format element must not be the first format element that is not a prefix format element. There can be any number of comma or G format elements.
- Blanks must not be specified between format elements. Leading and trailing blanks can be specified but are ignored when formatting the result.

Table 54. Format elements for the VARCHAR_FORMAT (Numeric to VARCHAR) function

Format element	Description
0	Each 0 represents a significant digit. Leading zeros in a number are displayed as zeros.
9	Each 9 represents a significant digit. Leading zeros in a number are displayed as blanks. Only group separators that have at least one digit to the left of the separator are generated.
S	Prefix: If *numeric-expression* is a negative number, a leading minus sign (-) is included in the result. If *numeric-expression* is a positive number, a leading plus sign (+) is included in the result.
$	Prefix: A leading dollar sign ($) is included in the result.
MI	Suffix: If *numeric-expression* is a negative number, a trailing minus sign (-) is included in the result. If *numeric-expression* is a positive number, a trailing blank is included in the result.
PR	Suffix: If *numeric-expression* is a negative number, a leading less than character (<) and a trailing greater than character (>) are included in the result. If *numeric-expression* is a positive number, a leading space and a trailing space are included in the result.
,	Specifies that a comma be included in that location in the result. This comma is used as a group separator.
.	Specifies that a period be included in that location in the result. This period is used as a decimal point.
L	Prefix or Suffix: Specifies that the local currency symbol be included in that location in the result. The currency symbol is retrieved from message CPX8416 in message file QCPFMSG in library *LIBL.
D	Specifies that the local decimal point character be included in that location in the result. The decimal character is retrieved from message CPX8416 in message file QCPFMSG in library *LIBL.

Table 54. Format elements for the VARCHAR_FORMAT (Numeric to VARCHAR) function (continued)

Format element	Description
G	Specifies that the local group separator character be included in that location in the result. If the local decimal character as retrieved from message CPX8416 in message file QCPFMSG in library *LIBL is a period, the group separator will be a comma. If the local decimal character is a comma, the group separator will be a period.

If *format-string* is not specified, the function is equivalent to VARCHAR(*numeric-expression*).

The result is a representation of the *numeric-expression* value (which might be rounded) in the format that is specified by *format-string*. Prior to being formatted, the value of *numeric-expression* is rounded by using the ROUND function if the number of digits to the right of the decimal point is greater than the number of digit format elements ('0' or '9') to the right of the decimal point in *format-string*. *format-string* is applied to this *rounded-input-value* according to the following rules:

- The result does not include any digit characters to the left of the decimal point if all of the following conditions are true:
  - -1 < *rounded-input-value* < 1
  - *format-string* does not include a '0' format element to the left of the decimal point
  - *format-string* includes at least one digit format element ('0' or '9') to the right of the decimal point
- The result includes a single 0 character immediately before the implicit or explicit decimal point if all of the following conditions are true:
  - The value of *rounded-input-value* is 0 or -0
  - *format-string* includes only the '9' digit format elements to the left of the implicit or explicit decimal point
  - *format-string* does not include any digit format elements to the right of the decimal point
- If *format-string* includes both '0' and '9' format elements to the left of the decimal point, the position of the first digit format element from the left side of the format string determines the presence of leading blanks or zeroes. All '9' format elements specified after the leftmost '0' format element to the left of the implicit or explicit decimal point are treated the same as if a '0' format element had been specified. For example, the *format-string* value '99099' is the same as the value '99000'.
- If the number of digits to the right of the decimal point in *rounded-input-value* is less than the number of digit format elements to the right of the decimal point in *format-string*, the result includes the number of digit characters to the right of the decimal point that corresponds to the number of digit format elements to the right of the decimal point in *format-string*, padded to the right with zeros.
- If the number of digits to the left of the decimal point in *rounded-input-value* is greater than the number of digit format elements to the left of the decimal point in *format-string*, the result is a string of number sign (#) characters that matches the length that *format-string* produces in the result for valid values.
- If the value of *rounded-input-value* represents any of the positive or negative special values, Infinity, sNaN, or NaN, the string 'INFINITY', 'SNAN', 'NAN', '-INFINITY', '-SNAN', or '-NAN' is returned without using the format that is

specified by *format-string*. The decimal floating-point special value sNaN does not result in an exception when converted to a string.

- If *format-string* does not include any of the sign format elements 'S', "MI', or 'PR', and the value of *rounded-input-value* is negative, a minus sign (–) is included in the result. Otherwise, a blank is included in the resulting string. The minus sign or blank immediately precedes the first digit of the result to the left of the decimal point, or the decimal point if there are no digits to the left of the decimal point.

The result is a varying-length character string representation of *rounded-input-value*. If a single argument is specified the length attribute is 42. Otherwise the length attribute is 254. The actual length of the result is determined by *format-string*, if specified. Otherwise, the actual length of the result is the smallest number of characters that can represent the value of *rounded-input-value*. If the resulting string exceeds the length attribute of the result, the result will be truncated.

The CCSID of the result is the same as the CCSID of the *format-string*. If *format-string* is not specified, the CCSID of the result is the default SBCS CCSID at the current server.

Note

Syntax alternatives: TO_CHAR is a synonym for VARCHAR_FORMAT.

Examples

Example: Timestamp to VARCHAR

- Set the character variable *TVAR* to a string representation of the timestamp value of RECEIVED from CORPDATA.IN_TRAY, formatted as YYYY-MM-DD HH24:MI:SS.

```
SELECT VARCHAR_FORMAT(RECEIVED,'YYYY-MM-DD HH24:MI:SS')
  INTO :TVAR
  FROM CORPDATA.IN_TRAY
  WHERE SOURCE = 'CHAAS'
```

Returns the string:

```
1988-12-22 14:07:21
```

Assuming that the value in the RECEIVED column is one second before the beginning of the year 2000 (December 31, 1999 at 23:59:59pm), the following string is returned:

```
1999-12-31 23:59:59
```

The result would be different if HH12 had been specified instead of HH24 in the format string:

```
1999-12-31 11:59:59
```

Example: Timestamp to VARCHAR

- Assume that the variable *TMSTAMP* is defined as a TIMESTAMP and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(255).

```
Function invocation                          Result
-------------------                          ------
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHHMISS')     20070309020738
```

```
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHH24MISS')   20070309140738
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHHMI')       200703090207
VARCHAR_FORMAT(TMSTAMP,'DD/MM/YY')           09/03/07
VARCHAR_FORMAT(TMSTAMP,'MM-DD-YYYY')         03-09-2007
VARCHAR_FORMAT(TMSTAMP,'J')                  2454169
VARCHAR_FORMAT(TMSTAMP,'Q')                  1
VARCHAR_FORMAT(TMSTAMP,'W')                  2
VARCHAR_FORMAT(TMSTAMP,'IW')                 10
VARCHAR_FORMAT(TMSTAMP,'WW')                 10
VARCHAR_FORMAT(TMSTAMP,'Month')              March
VARCHAR_FORMAT(TMSTAMP,'MONTH')              MARCH
VARCHAR_FORMAT(TMSTAMP,'MON')                MAR
```

Example: Timestamp to VARCHAR

- Assume that the variable *DTE* is defined as a DATE and has the value of '2007-03-09'. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(255):

```
Function invocation                      Result
-------------------                      ------
VARCHAR_FORMAT(DTE,'YYYYMMDD')           20070309
VARCHAR_FORMAT(DTE,'YYYYMMDDHH24MISS')   20070309000000
```

Example: Timestamp to VARCHAR

- Format the hour of the specified string representation of a timestamp:

```
SELECT
  VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH'),
  VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH12')
  VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH24'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH12'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH12'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH12'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH24'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH24'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH24')
FROM SYSIBM.SYSDUMMY1;
```

The previous SELECT statement returns the following values:

```
'02'  '02'  '14'  '12'  '12'  '12'  '12'  '12'  '12'  '00'  '12'  '24'
```

Example: Numeric to VARCHAR

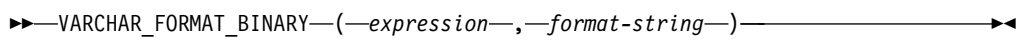
- Assume that the variables POSNUM and NEGNUM are defined as DECFLOAT(34) and have the following values: '1234.56' and '-1234.56', respectively. The following examples show several invocations of the function and the resulting string values.

```
Function invocation                      Result
-------------------------                -----------
VARCHAR_FORMAT(POSNUM)                   '1234.56'
VARCHAR_FORMAT(NEGNUM)                   '-1234.56'

VARCHAR_FORMAT(POSNUM,'9999.99')         ' 1234.56'
VARCHAR_FORMAT(NEGNUM,'9999.99')         '-1234.56'

VARCHAR_FORMAT(POSNUM,'99999.99')        '  1234.56'
VARCHAR_FORMAT(NEGNUM,'99999.99')        ' -1234.56'

VARCHAR_FORMAT(POSNUM,'00000.00')        ' 01234.56'
VARCHAR_FORMAT(NEGNUM,'00000.00')        '-01234.56'
```

```
|
|                       VARCHAR_FORMAT(POSNUM,'9999.99MI')        '1234.56 '
|                       VARCHAR_FORMAT(NEGNUM,'9999.99MI')        '1234.56-'
|
|                       VARCHAR_FORMAT(POSNUM,'S9999.99')         '+1234.56'
|                       VARCHAR_FORMAT(NEGNUM,'S9999.99')         '-1234.56'
|
|                       VARCHAR_FORMAT(POSNUM,'9999.99PR')        ' 1234.56 '
|                       VARCHAR_FORMAT(NEGNUM,'9999.99PR')        '<1234.56>'
|
|                       VARCHAR_FORMAT(POSNUM,'S$9,999.99')       '+$1,234.56'
|                       VARCHAR_FORMAT(NEGNUM,'S$9,999.99')       '-$1,234.56'
```

VARCHAR_FORMAT_BINARY

The VARCHAR_FORMAT_BINARY function returns a character string representation of a bit string that has been formatted using a *format-string*.



expression
An expression that returns a built-in binary string or character FOR BIT DATA string. The length of *expression* must be equal to the number of 'x' or 'X' characters in the *format-string* divided by 2.

format-string
An expression that returns a built-in character string or graphic string data type. *format-string* contains a template for how the value for *expression* is to be formatted.

Valid format strings are: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' and 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' where each 'x' or 'X' corresponds to one hexadecimal digit from *expression*. If 'x' is specified, the character returned for the corresponding hexadecimal digit will be lower case. Otherwise, the character returned will be upper case.

The result of the function is a varying-length character string with the length attribute and actual length based on the format string. For the two valid format strings, the length attribute and actual length of the result is 36. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The CCSID of the result is the CCSID of *format-string*.

Note

Syntax alternatives: VARCHAR_FORMAT_BIT is a synonym for VARCHAR_FORMAT_BINARY.

Example

- Represent a Universal Unique Identifier in its formatted form:

```
VALUES VARCHAR_FORMAT_BINARY(BX'd83d6360181811db9804b622a1ef5492',
                             'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
```

  Result returned:

```
'd83d6360-1818-11db-9804-b622a1ef5492'
```

- Represent a Universal Unique Identifier in its formatted form:

```
VALUES VARCHAR_FORMAT_BINARY(BX'd83d6360181811db9804b622a1ef5492',
                             'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

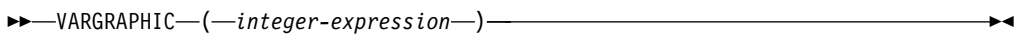
  Result returned:

```
'D83D6360-1818-11DB-9804-B622A1EF5492'
```

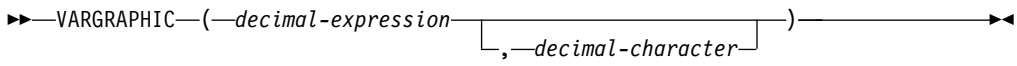
VARGRAPHIC

The VARGRAPHIC function returns a graphic-string representation.

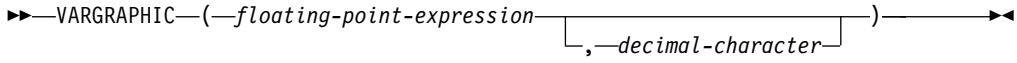
Integer to Vargraphic



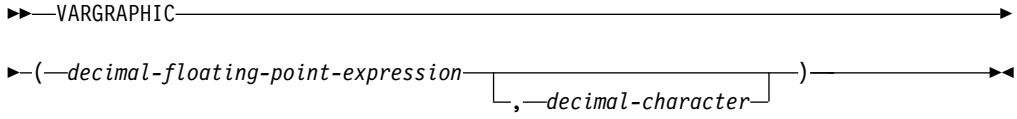
Decimal to Vargraphic



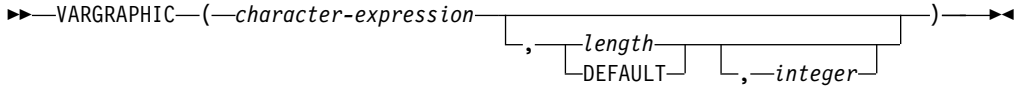
Floating-point to Vargraphic



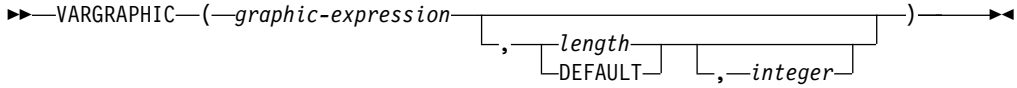
Decimal floating-point to Vargraphic



Character to Vargraphic



Graphic to Vargraphic



The VARGRAPHIC function returns a graphic-string representation of

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT.
- A decimal number if the first argument is a packed or zoned decimal number.
- A double-precision floating-point number if the first argument is a DOUBLE or REAL.
- A decimal floating-point number if the first argument is DECFLOAT.
- A character string if the first argument is any type of character string.
- A graphic string if the first argument is a Unicode graphic string.

The result of the function is a varying-length graphic string (VARGRAPHIC).

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value. If the first argument is an empty string or the EBCDIC string X'0E0F', the result is an empty string.

Integer to Vargraphic

integer-expression
An expression that returns a value that is an integer data type (either SMALLINT, INTEGER, or BIGINT).

The result is a varying-length graphic string of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. It is left justified.

- If the argument is a small integer, the length attribute of the result is 6.
- If the argument is a large integer, the length attribute of the result is 11.
- If the argument is a big integer, the length attribute of the result is 20.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit or the *decimal-character*.

The CCSID of the result is 1200 (UTF-16).

Decimal to Vargraphic

decimal-expression
An expression that returns a value that is a packed or zoned decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is wanted, the DECIMAL scalar function can be used to make the change.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length graphic string representation of the argument. The result includes a decimal character and up to *p* digits, where *p* is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length attribute of the result is 2+*p* where *p* is the precision of the *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing characters are included. Leading zeros are not included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit or the *decimal-character*.

The CCSID of the result is 1200 (UTF-16).

Floating-point to Vargraphic

floating-point expression
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length graphic string representation of the argument in the form of a floating-point constant.

The length attribute of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0E0.

The CCSID of the result is 1200 (UTF-16).

Decimal floating-point to Vargraphic

decimal-floating-point expression
An expression that returns a value that is a decimal floating-point data type.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see "Decimal point" on page 127.

The result is a varying-length graphic string representation of the argument in the form of a decimal floating-point constant.

The length attribute of the result is 42. The actual length of the result is the smallest number of characters that represents the value of the argument, including the sign, digits, and *decimal-character*. Trailing zeros are significant. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the *decimal-character*. If the argument is zero, the result is 0.

If the DECFLOAT value is Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, a minus sign will be the first character in the string. The DECFLOAT special value sNaN does not result in an exception when converted to a string.

The CCSID of the result is 1200 (UTF-16).

Character to Vargraphic

character-expression
Specifies a character string expression. It cannot be a CHAR or VARCHAR bit data.

length
An integer constant that specifies the length attribute of the result and must be an integer constant between 1 and 16370 if the first argument is not nullable or between 1 and 16369 if the first argument is nullable.

If the second argument is not specified, or if DEFAULT is specified, the length attribute of the result is the same as the length attribute of the first argument, except if the expression is an empty string or the EBCDIC string X'0E0F', the length attribute of the result is 1.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. Each character of the argument determines a character of the result. If the length (in characters) of the *character-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID of the result. It must be a DBCS, UTF-16, or UCS-2 CCSID. The CCSID cannot be 65535. If the CCSID represents Unicode graphic data, each character of the argument determines a character of the result. The nth character of the result is the UTF-16 or UCS-2 equivalent of the nth character of the argument.

If *integer* is not specified then the CCSID of the result is determined by a mixed CCSID. Let M denote that mixed CCSID.

In the following rules, S denotes one of the following:

- If the string expression is a host variable containing data in a foreign encoding scheme, S is the result of the expression after converting the data to a CCSID in a native encoding scheme. (See "Character conversion" on page 32 for more information.)
- If the string expression is data in a native encoding scheme, S is that string expression.

M is determined as follows:

- If the CCSID of S is 1208 (UTF-8), M is 1200 (UTF–16).
- If the CCSID of S is a mixed CCSID, M is that CCSID.
- If the CCSID of S is an SBCS CCSID:
  - If the CCSID of S has an associated mixed CCSID, M is that CCSID.
  - Otherwise the operation is not allowed.

The following table summarizes the result CCSID based on M.

M	Result CCSID	Description	DBCS Substitution Character
930	300	Japanese EBCDIC	X'FEFE'
933	834	Korean EBCDIC	X'FEFE'
935	837	S-Chinese EBCDIC	X'FEFE'
937	835	T-Chinese EBCDIC	X'FEFE'
939	300	Japanese EBCDIC	X'FEFE'
5026	4396	Japanese EBCDIC	X'FEFE'
5035	4396	Japanese EBCDIC	X'FEFE'

The equivalence of SBCS and DBCS characters depends on M. Regardless of the CCSID, every double-byte code point in the argument is considered a DBCS character, and every single-byte code point in the argument is considered an SBCS character with the exception of the EBCDIC mixed data shift codes `X'0E'` and `X'0F'`.

- If the nth character of the argument is a DBCS character, the nth character of the result is that DBCS character.
- If the nth character of the argument is an SBCS character that has an equivalent DBCS character, the nth character of the result is that equivalent DBCS character.
- If the nth character of the argument is an SBCS character that does not have an equivalent DBCS character, the nth character of the result is the DBCS substitution character.

Graphic to Vargraphic

graphic-expression
An expression that returns a value that is a graphic string.

length
An integer constant that specifies the length attribute of the result and must be an integer constant between 1 and 16370 if the first argument is not nullable or between 1 and 16369 if the first argument is nullable.

If the second argument is not specified, or if DEFAULT is specified, the length attribute of the result is the same as the length attribute of the first argument, except if the expression is an empty string, the length attribute of the result is 1.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. Each character of the argument determines a character of the result. If the length (in characters) of the *graphic-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer
An integer constant that specifies the CCSID of the result. It must be a DBCS, UTF-16, or UCS-2 CCSID. The CCSID cannot be 65535.

If *integer* is not specified then the CCSID of the result is the CCSID of the first argument.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the first argument is a string and the length attribute is specified. For more information, see "CAST specification" on page 185.

Example

- Using the EMPLOYEE table, set the host variable VAR_DESC (VARGRAPHIC(24)) to the VARGRAPHIC equivalent of the first name (FIRSTNME) for employee number (EMPNO) '000050'.

```
SELECT VARGRAPHIC(FIRSTNME)
  INTO :VAR_DESC
  FROM EMPLOYEE
  WHERE EMPNO = '000050'
```

WEEK

The WEEK function returns an integer between 1 and 54 that represents the week of the year. The week starts with Sunday, and January 1 is always in the first week.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Using the PROJECT table, set the host variable WEEK (INTEGER) to the week that project ('PL2100') ended.

```
SELECT WEEK(PRENDATE)
  INTO :WEEK
  FROM PROJECT
  WHERE PROJNO = 'PL2100'
```

  Results in WEEK being set to 38.

- Assume that table X has a DATE column called DATE_1 with various dates from the list below.

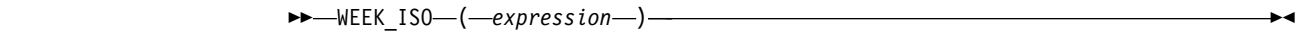
```
SELECT DATE_1, WEEK(DATE_1)
  FROM X
```

  Results in the following list shows what is returned by the WEEK function for various dates.

```
1997-12-28    53
1997-12-31    53
1998-01-01    1
1999-01-01    1
1999-01-04    2
1999-12-31    53
2000-01-01    1
2000-01-03    2
```

WEEK_ISO

The WEEK_ISO function returns an integer between 1 and 53 that represents the week of the year. The week starts with Monday. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4. Thus, it is possible to have up to 3 days at the beginning of the year appear as the last week of the previous year or to have up to 3 days at the end of a year appear as the first week of the next year.



expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Using the PROJECT table, set the host variable WEEK (INTEGER) to the week that project ('AD2100') ended.

```
SELECT WEEK_ISO(PRENDATE)
  INTO :WEEK
  FROM PROJECT
  WHERE PROJNO = 'AD3100'
```

  Results in WEEK being set to 5.

- Assume that table X has a DATE column called DATE_1 with various dates from the list below.

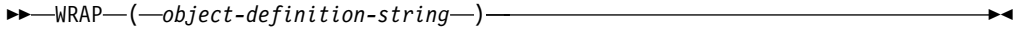
```
SELECT DATE_1, WEEK_ISO(DATE_1)
  FROM X
```

  Results in the following:

```
1997-12-28    52
1997-12-31    1
1998-01-01    1
1999-01-01    53
1999-01-04    1
1999-12-31    52
2000-01-01    52
2000-01-03    1
```

WRAP

The WRAP function transforms a readable DDL statement into an obfuscated DDL statement.



In an obfuscated DDL statement, the procedural logic and embedded SQL statements are scrambled in such a way that any intellectual property in the logic cannot be easily extracted.

The schema is SYSIBMADM.

object-definition-string
A string of type CLOB containing a DDL statement. It can be one of the following SQL statements:

- CREATE FUNCTION (SQL scalar)
- CREATE FUNCTION (SQL table)
- CREATE PROCEDURE (SQL)

The result is a string of type CLOB(2M) which contains an encoded version of the input statement. The result cannot be null. The encoding consists of a prefix of the original statement up to and including the routine signature or trigger name, followed by the keyword WRAPPED. This keyword is followed by information about the application server that invoked the function. The information has the form *pppvvrrm* where:

- *ppp* identifies the product using the following 3 characters:
  - QSQ for DB2 for i
  - SQL for DB2 Database for Linux, UNIX, and Windows
- *vv* is a two-digit version identifier, such as '09'
- *rr* is a two-digit release identifier, such as '07'
- *m* is a one-character modification level identifier, such as '0'

For example DB2 for i version 7.1 is identified as 'QSQ07010'.

This application server information is followed by a string of letters (a-z and A-Z), digits (0-9), underscores, and colons.

The encoded DDL statement may be up to one-third longer than the plain text form of the statement. If the result exceeds the maximum length for SQL statements, an error is issued (SQLSTATE 54001).

Note

The encoding of the statement is meant to obfuscate the content and should not be considered as a form of strong encryption.

Example

Produce an obfuscated version of a function that computes a yearly salary from an hourly wage given a 40 hour work week.

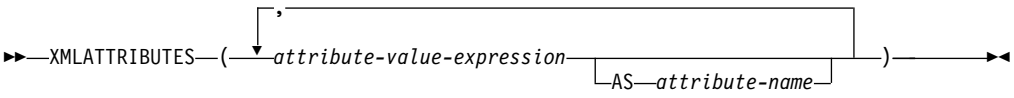
```
VALUES WRAP('CREATE FUNCTION salary(wage DECFLOAT) RETURNS DECFLOAT
                      RETURN wage * 40 * 52')
```

The result of this statement would be something of the form:

```
CREATE FUNCTION salary(wage DECFLOAT) WRAPPED QSQ07010 <encoded-suffix>
```

XMLATTRIBUTES

The XMLATTRIBUTES function constructs XML attributes from the arguments.



This function can only be used as an argument of the XMLELEMENT function. The result is an XML sequence containing an XML attribute for each non-null *attribute-value-expression* argument.

attribute-value-expression
An expression whose result is the attribute value. The data type of *attribute-value-expression* must not be ROWID, DATALINK, XML or a distinct type that is based on ROWID, DATALINK, or XML. The expression can be any SQL expression. If the expression is not a simple column reference, an attribute name must be specified.

attribute-name
Specifies an attribute name. The name is an SQL identifier that must be in the form of an XML qualified name, or QName. See the W3C XML namespace specifications for more details on valid names. The attribute name cannot be "xmlns" or prefixed with "xmlns:". A namespace is declared using the function XMLNAMESPACES. Duplicate attribute names, whether implicit or explicit, are not allowed.

If *attribute-name* is not specified, *attribute-value-expression* must be a column name. The attribute name is created from the column name using the fully escaped mapping from a column name to an XML attribute name.

The result of the function is XML. If the result of any *attribute-value-expression* can be null, the result can be null; if the result of every *attribute-value-expression* is null, the result is the null value.

Example

Note: XMLATTRIBUTES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Produce an element with attributes.

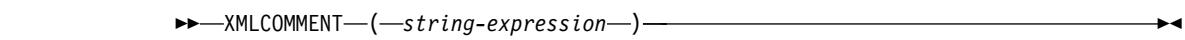
```
SELECT E.EMPNO, XMLELEMENT(
  NAME "Emp",
  XMLATTRIBUTES(
   E.EMPNO, E.FIRSTNME ||' '|| E.LASTNAME AS "name"
    )
   )
 AS "Result"
  FROM EMPLOYEE E
  WHERE E.EDLEVEL = 12
```

This query produces the following result:

```
EMPNO   Result
000290  <Emp EMPNO="000290" name="JOHN PARKER"/>
000310  <Emp EMPNO="000310" name="MAUDE SETRIGHT"/>
200310  <Emp EMPNO="200310" name="MICHELLE SPRINGER"/>
```

XMLCOMMENT

The XMLCOMMENT function returns an XML value with the input argument as the content.



string-expression
An expression that returns a value of any built-in character-string or graphic-string data type. It cannot be CHAR or VARCHAR bit data. The result of *string-expression* is parsed to check for conformance to the content of an XML comment, as specified by the following rules:

- Two adjacent hyphens ('--') must not occur in the string expression.
- The string expression must not end with a hyphen ('-').
- Each character of the string can be any Unicode character, excluding the surrogate blocks, X'FFFE', and X'FFFF'.[76]

If *string-expression* does not conform to the previous rules, an error is returned.

The result of the function is XML. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Generate an XML comment.

```
SELECT XMLCOMMENT('This is an XML comment')
  FROM SYSIBM.SYSDUMMY1
```

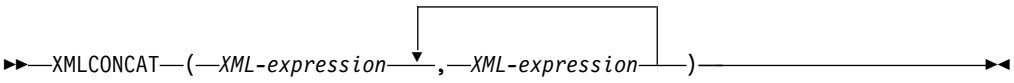
This query produces the following result:

```
<!--This is an XML comment-->
```

76. Valid Unicode characters consist of the following Unicode code points: #x9, #xA, #xD, #x20-#xD7FF, #xE000-#xFFFD, #x10000-#x10FFFF.

XMLCONCAT

The XMLCONCAT function returns a sequence containing the concatenation of a variable number of XML input arguments.



XML-expression
An expression that returns an XML value.

The result of the function is an XML sequence that contains the concatenation of the non-null input XML values. Null values in the input are ignored.

The result of the function is XML. The result can be null; if the result of every input value is null, the result is the null value.

Examples

Note: XMLCONCAT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Concatenate first name and last name elements by using "first" and "last" element names for each employee

```
SELECT XMLSERIALIZE(
      XMLCONCAT(
        XMLELEMENT(NAME "first", e.firstnme),
        XMLELEMENT(NAME "last", e.lastname)
       ) AS VARCHAR(100) ) AS "result"
FROM EMPLOYEE E
WHERE e.lastname = 'SMITH'
```

  The result of the query would look similar to the following result:

```
result
----------------------------------------
<first>DANIEL</first><last>SMITH</last>
<first>PHILIP</first><last>SMITH</last>
```

- Construct a department element for departments A00 and B01 containing a list of employees sorted by first name. Include an introductory comment immediately preceding the department element.

```
SELECT XMLCONCAT(
  XMLCOMMENT (
   'Confirm these employees are on track for their product schedule'),
  XMLELEMENT(
     NAME "Department",
     XMLATTRIBUTES( E.WORKDEPT AS "name"),
     XMLAGG(
       XMLELEMENT(NAME "emp", E.FIRSTNME)
       ORDER BY E.FIRSTNME)
           ))
         FROM EMPLOYEE E
  WHERE E.WORKDEPT IN ('A00', 'B01')
GROUP BY E.WORKDEPT
```

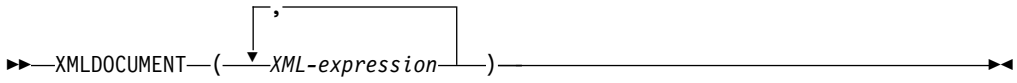
  This query produces the following result:

```
<!--Confirm these employees are on track for their product schedule-->
<Department name="A00">
<emp>CHRISTINE</emp>
<emp>SEAN</emp>
<emp>VINCENZO</emp>
```

```
                        </Department>
                        <!--Confirm these employees are on track for their product schedule-->
                        <Department name="B01">
                        <emp>MICHAEL</emp>
                        </Department>
```

XMLDOCUMENT

The XMLDOCUMENT function returns an XML value.



XML-expression
An expression that returns an XML value.

The result of the function is XML. If the result of *XML-expression* can be null, the result can be null; if every *XML-expression* is null, the result is the null value.

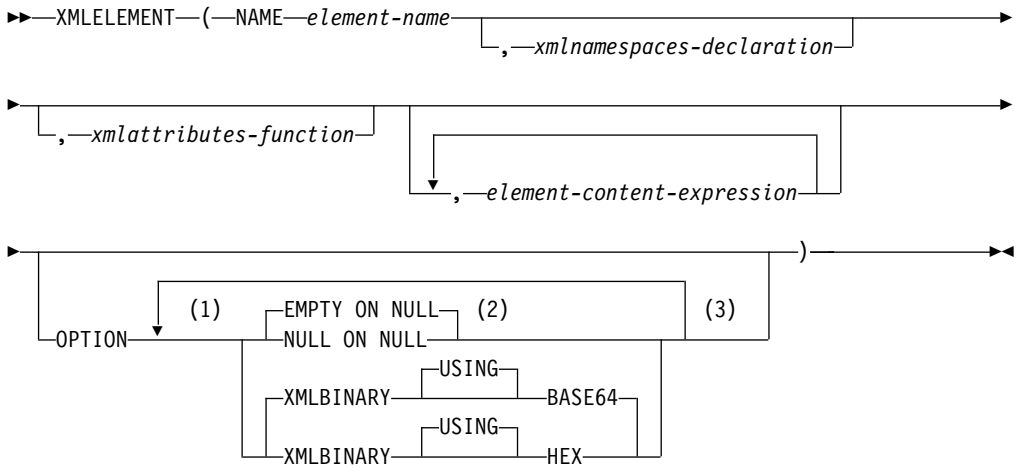
Example

- Insert a constructed document into an XML column.

```
INSERT INTO T1 VALUES(123,
            (SELECT XMLDOCUMENT(
                 XMLELEMENT(NAME "Emp",
                                  E.FIRSTNME || ' ' || E.LASTNAME,
                            XMLCOMMENT('This is just a simple example')
                           ))
                FROM EMPLOYEE E
                WHERE E.EMPNO = '000120'))
```

XMLELEMENT

The XMLELEMENT function returns an XML value that is an XML element.



Notes:

1 The OPTION clause can only be specified if at least one *xmlattributes-function* or *element-content-expression* is specified

2 If *element-content-expression* is not specified, EMPTY ON NULL or NULL ON NULL must not be specified.

3 The same clause must not be specified more than once.

NAME *element-name*
Specifies the name of an XML element. The name is an SQL identifier that must be in the form of an XML qualified name, or QName. See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope.

xmlnamespaces-declaration
Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLELEMENT function. The namespaces apply to any nested XML functions within the XMLELEMENT function, regardless of whether or not they appear inside another subselect. See "XMLNAMESPACES" on page 568 for more information on declaring XML namespaces.

If *xmlnamespaces-declaration* is not specified, namespace declarations are not associated with the constructed element.

xmlattributes-function
Specifies the XML attributes for the element. The attributes are the result of the XMLATTRIBUTES function. See "XMLATTRIBUTES" on page 556 for more information on construction attributes.

element-content-expression
The content of the generated XML element node is specified by an expression or a list of expressions. The expression can be any SQL expression of any SQL data type except for ROWID or DATALINK. The expression is used to construct the namespace declarations, attributes, and content of the constructed element.

If *element-content-expression* is not specified, an empty string is used as the content for the element and NULL ON NULL or EMPTY ON NULL must not be specified.

OPTION
Specifies additional options for constructing the XML element. This clause has no impact on nested XMLELEMENT invocations specified in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL
Specifies whether a null value or an empty element is to be returned if the value of every *element-content-expression* is the null value. This option only affects null handling of element contents, not attribute values. The default is EMPTY ON NULL.

EMPTY ON NULL
If the value of each *element-content-expression* is null, an empty element is returned.

NULL ON NULL
If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX
Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

XMLBINARY USING BASE64
Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+', and '/') to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING HEX
Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hexadecimal encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an element name, an optional collection of namespace declarations, an optional collection of attributes, and zero or more arguments that make up the content of the XML element. The result is an XML sequence containing an XML element node or the null value. If the results of all *element-content-expression* arguments are empty strings, the result is an XML sequence that contains an empty element.

The result of the function is XML. The result can be null; if all the *element-content-expression* argument values are null and the NULL ON NULL option is in effect, the result is the null value.

Rules about using namespaces within XMLELEMENT: The following rules describe scoping of namespaces:

- The namespaces declared in the XMLNAMESPACES declaration are the in-scope namespaces of the element constructed by the XMLELEMENT function. If the element is serialized, then each of its in-scope namespaces will be serialized as a namespace attribute unless it is an in-scope namespace of an XML value that includes this element.
- The scope of these namespaces is the lexical scope of the XMLELEMENT function, including the element name, the attribute names that are specified in the XMLATTRIBUTES function, and all *element-content-expression*s. These are used to resolve the QNames in the scope.
- If an attribute of the constructed element comes from an *element-content-expression*, its namespace might not already by declared as an in-scope namespace of the constructed element. In this case, a new namespace is created for it. If this would result in a conflict, which means that the prefix of the attribute name is already bound to a different URI by an in-scope namespace, DB2 generates a different prefix to be used in the attribute name. A namespace is created for this generated prefix. The name of the generated prefix follows the following pattern: db2ns-*xx*, where *xx* is a pair of characters chosen from the set [A-Z, a-z, 0-9].

Example

Note: XMLELEMENT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

The following examples use a temporary CANDIDATES employee table:

```
DECLARE GLOBAL TEMPORARY TABLE CANDIDATES
  (EMPNO CHAR(6),
   FIRSTNME VARCHAR(12),
   MIDINIT CHAR(1),
   LASTNAME VARCHAR(15),
   WORKDEPT CHAR(4),
   EDLEVEL INT)
INSERT INTO SESSION.CANDIDATES
   VALUES('A0001', 'John', 'A', 'Parker', 'X001', 12)
INSERT INTO SESSION.CANDIDATES
   VALUES('B0001', NULL, NULL, 'Smith', 'X001', 12)
INSERT INTO SESSION.CANDIDATES
   VALUES('B0002', NULL, NULL, NULL, 'X001', NULL)
INSERT INTO SESSION.CANDIDATES
   VALUES(NULL, NULL, NULL, NULL, 'X001', NULL)
```

- The following statement used the XMLELEMENT function to create an XML element that contains an employee's name. The statement also sets the employee number as an attribute names serial. If there is a null value in the referenced column, the function returns the null value:

```
SELECT E.EMPNO, E.FIRSTNME, E.LASTNAME,
       XMLELEMENT(NAME "foo:Emp",
                  XMLNAMESPACES('http://www.foo.com' AS "foo"),
                  XMLATTRIBUTES(E.EMPNO AS "serial"),
                  E.FIRSTNME, E.LASTNAME
                  OPTION NULL ON NULL) AS "Result"
FROM SESSION.CANDIDATES E
```

  This query produces the following result:

```
EMPNO  FIRSTNME   LASTNAME  Result
------ ---------  --------  -------------------
A0001  John       Parker    <foo:Emp xmlns:foo="http://www.foo.com"
                                serial="A0001">JohnParker</foo:Emp>
```

```
B0001  (null)     Smith     <foo:Emp xmlns:foo="http://www.foo.com"
                                serial="B0001">Smith</foo:Emp>
B0002  (null)     (null)    (null)
(null) (null)     (null)    (null)
```

- The following example is similar to the previous one. However, when there is a null value in the referenced column, an empty element is returned:

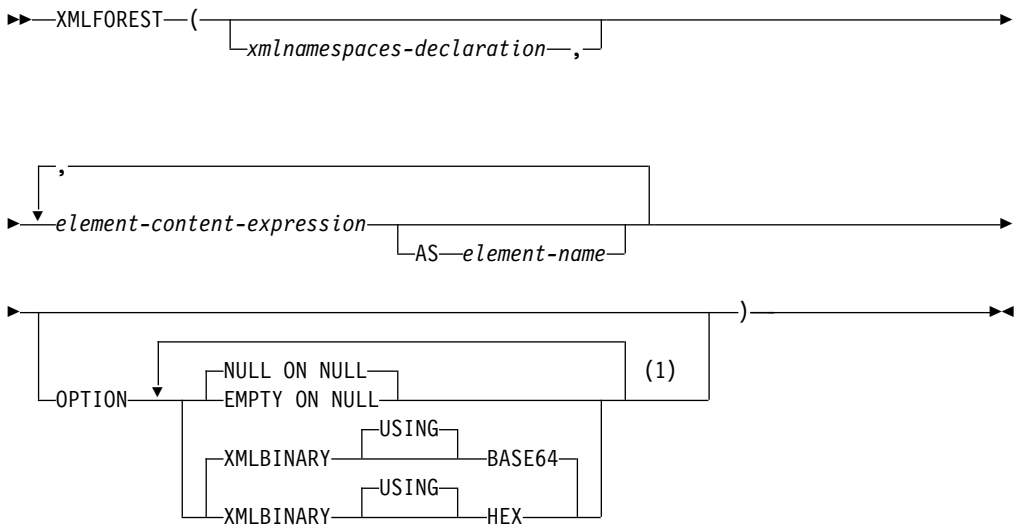
```
SELECT E.EMPNO, E.FIRSTNME, E.LASTNAME,
       XMLELEMENT(NAME "foo:Emp",
                  XMLNAMESPACES('http://www.foo.com' AS "foo"),
                  XMLATTRIBUTES(E.EMPNO AS "serial"),
                  E.FIRSTNME, E.LASTNAME
                  OPTION EMPTY ON NULL) AS "Result"
FROM SESSION.CANDIDATES E
```

  This query produces the following result:

```
EMPNO  FIRSTNME   LASTNAME  Result
------ ---------  --------  ------------------
A0001  John       Parker    <foo:Emp xmlns:foo="http://www.foo.com"
                                serial="A0001">JohnParker</foo:Emp>
B0001  (null)     Smith     <foo:Emp xmlns:foo="http://www.foo.com"
                                serial="B0001">Smith</foo:Emp>
B0002  (null)     (null)    <foo:Emp xmlns:foo="http://www.foo.com"
                                serial="B0002"></foo:Emp>
(null) (null)     (null)    <foo:Emp></foo:Emp>
```

XMLFOREST

The XMLFOREST function returns an XML value that is a sequence of XML elements.



Notes:

1 The same clause must not be specified more than once.

xmlnamespace-declaration
Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLFOREST function. The namespaces apply to any nested XML functions within the XMLFOREST function, regardless of whether or not they appear inside another subselect. See "XMLNAMESPACES" on page 568 for more information on declaring XML namespaces.

If *xmlnamespace-declaration* is not specified, namespace declarations are not associated with the constructed XML elements.

element-content-expression
Specifies an expression that returns a value that is used for the content of a generated XML element. The data type of the expression must not be ROWID or DATALINK. If the expression is not a simple column reference, *element-name* must be specified.

AS *element-name*
Specifies an identifier that is used for the XML element name.

An XML element name must be an XML qualified name, or QName. See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope.

If *element-name* is not specified, *element-content-expression* must be a column name. The element name is created from the column name using the fully escaped mapping from a column name to a QName.

OPTION
Specifies options for the result for NULL values, binary data, and bit data. The options will not be inherited by XMLELEMENT or XMLFOREST functions that appear in *element-content-expression*.

NULL ON NULL or EMPTY ON NULL
Specifies whether a null value or an empty element is to be returned if the value of every *element-content-expression* is the null value. This option only affects null handling of the *element-content-expression* arguments. The default is NULL ON NULL.

NULL ON NULL
If the value of each *element-content-expression* is null, a null value is returned.

EMPTY ON NULL
If the value of each *element-content-expression* is null, an empty element is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX
Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, ROWID, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

XMLBINARY USING BASE64
Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+', and '/') to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING HEX
Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hexadecimal encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

The result of the function is an XML value. If the result of any *element-content-expression* can be null, the result can be null; if the result of every *element-content-expression* is null and the NULL ON NULL option is in effect, the result is the null value.

The XMLFOREST function can be expressed by using XMLCONCAT and XMLELEMENT. For example, the following two expressions are semantically equivalent.

```
XMLFOREST(xmlnamespaces-declaration, arg1 AS name1, arg2 AS name2, ...)

XMLCONCAT(XMLELEMENT(NAME name1, xmlnamespaces-declaration, arg1),
          XMLELEMENT(NAME name2, xmlnamespaces-declaration, arg2),
 ... )
```

When constructing elements that will be copied as content of another element that defines default namespaces, default namespaces should be explicitly undeclared in the copied element to avoid possible errors that could result from inheriting the

default namespace from the new parent element. Predefined namespace prefixes ('xs', 'xsi', 'xml', 'sqlxml') must also be declared explicitly when they are used.

Example

Note: XMLFOREST does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Construct a forest of elements with a default namespace.

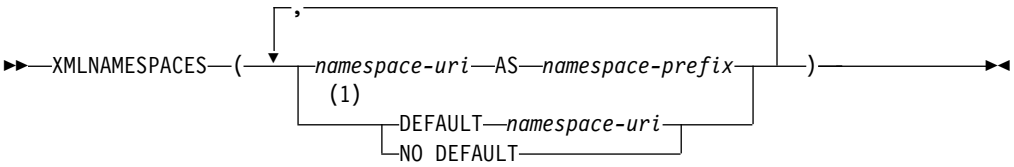
```
SELECT EMPNO,
       XMLFOREST(XMLNAMESPACES(DEFAULT 'http://hr.org',
                               'http://fed.gov' AS "d"),
                 LASTNAME, JOB AS "d:job") AS "Result"
FROM EMPLOYEE WHERE EDLEVEL = 12
```

  This query produces the following result:

```
EMPNO  Result
000290 <LASTNAME xmlns:"http://hr.org" xmlns:d="http://fed.gov">PARKER
       </LASTNAME>
       <d:job xmlns:"http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>
000310 <LASTNAME xmlns:"http://hr.org" xmlns:d="http://fed.gov">SETRIGHT
       </LASTNAME>
       <d:job xmlns:"http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>
200310 <LASTNAME xmlns:"http://hr.org" xmlns:d="http://fed.gov">SPRINGER
       </LASTNAME>
       <d:job xmlns:"http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>
```

XMLNAMESPACES

The XMLNAMESPACES declaration constructs namespace declarations from the arguments. This declaration can only be used as an argument for the XMLELEMENT and XMLFOREST functions. The result is one or more XML namespace declarations containing in-scope namespaces for each non-null input value.



Notes:

1 The DEFAULT or NO DEFAULT clause can only be specified one time.

namespace-uri
Specifies an SQL character string constant that contains the namespace name or a universal resource identifier (URI). The character string constant must not be an empty string if it is used with *namespace-prefix*.

AS *namespace-prefix*
Specifies a namespace prefix. The prefix is an SQL identifier that must be in the form of an XML NCName. See the W3C XML namespace specifications for more details on valid names. The prefix must not be "xml" or "xmlns". The prefix must be unique within the list of namespace declarations.

The following namespace prefixes are pre-defined in SQL/XML: "xml", "xs", "xsd", "xsi", and "sqlxml". Their bindings are:

- xmlns:xml = "http://www.w3.org/XML/1998/namespace"
- xmlns:xs = "http://www.w3.org/2001/XMLSchema"
- xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
- xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
- xmlns:sqlxml = "http:/standards.iso.org/iso/9075/2003/sqlxml"

DEFAULT *namespace-uri* **or NO DEFAULT**
Specifies whether a default namespace is to be used within the scope of this namespace declaration.

The scope of this namespace declaration is the specified XML element and all XML expressions that are contained in the specified XML element.

DEFAULT *namespace-uri*
Specifies the default namespace to use within the scope of this namespace declaration. The *namespace-uri* applies for unqualified names in the scope unless it is overridden in a nested scope by another DEFAULT declaration or by a NO DEFAULT declaration.

namespace-uri specifies an SQL character string constant that contains a namespace name or universal resource identifier (URI). The character string constant can be an empty string in the context of the DEFAULT clause.

NO DEFAULT
Specifies that no default namespace is to be used within the scope of this

namespace declaration. There is no default namespace in the scope unless the NO DEFAULT clause is overridden in a nested scope by a DEFAULT declaration.

The result of the function is an XML value that is an XML sequence that contains an XML namespace declaration for each specified namespace. The result cannot be null.

Examples

Note: XML processing does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Generate an "employee" element for each employee. The employee element is associated with XML namespace "urn:bo", which is bound to prefix "bo". The element contains attributes for names and a hiredate subelement.

```
SELECT E.EMPNO,
       XMLSERIALIZE(XMLELEMENT(NAME "bo:employee",
                               XMLNAMESPACES('urn:bo' AS "bo"),
                               XMLATTRIBUTES(E.LASTNAME, E.FIRSTNME),
                               XMLELEMENT(NAME "bo:hiredate", E.HIREDATE))
                    AS CLOB(200))
  FROM EMPLOYEE E WHERE E.EDLEVEL = 12
```

  This query produces the following result:

```
00029 <bo:employee xmlns:bo="urn:bo" LASTNAME="PARKER" FIRSTNME="JOHN">
      <bo:hiredate>1988-05-30</bo:hiredate>
      </bo:employee>
00031 <bo:employee xmlns:bo="urn:bo" LASTNAME="SETRIGHT" FIRSTNME="MAUDE">
      <bo:hiredate>1964-09-12</bo:hiredate>
      </bo:employee>
```

- Generate two elements for each employee using XMLFOREST. The first "lastname" element is associated with the default namespace "http://hr.org", and the second "job" element is associated with XML namespace "http://fed.gov", which is bound to prefix "d".

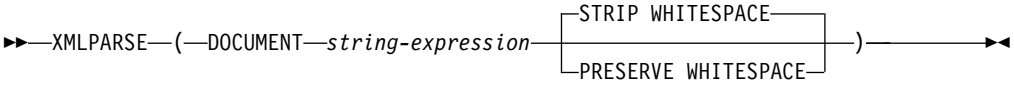
```
SELECT EMPNO,
       XMLSERIALIZE(XMLFOREST(XMLNAMESPACES(DEFAULT 'http://hr.org',
                                            'http://fed.gov' AS "d"),
                              LASTNAME, JOB AS "d:job")
                    AS CLOB(200))
  FROM EMPLOYEE WHERE EDLEVEL = 12
```

  This query produces the following result:

```
00029 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">PARKER
      </LASTNAME>
      <d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">
      OPERATOR</d:job>
00031 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">
      SETRIGHT</LASTNAME>
      <d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">
      OPERATOR</d:job>
```

XMLPARSE

The XMLPARSE function parses the arguments as an XML document and returns an XML value.



DOCUMENT
Specifies that the character string expression to be parsed must evaluate to a well-formed XML document that conforms to XML 1.0.

string-expression
An expression that returns a value that is a built-in character, Unicode graphic, or binary string. If a parameter marker is used, it must be explicitly cast to one of the supported data types.

STRIP WHITESPACE or PRESERVE WHITESPACE
Specifies whether or not whitespace in the input argument is to be preserved. If neither is specified, STRIP WHITESPACE is the default.

STRIP WHITESPACE
Specifies that whitespace characters will be stripped unless the nearest containing element has the attribute *xml:space='preserve'*. The whitespace characters in the CDATA section are also affected by this option.

PRESERVE WHITESPACE
Specifies that all whitespace is to be preserved, even when the nearest containing element has the attribute *xml:space='default'*.

The result of the function is XML. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value. The CCSID of the result is determined from *string-expression*. If *string-expression* has a CCSID of 65535, the value from the SQL_XML_DATA_CCSID QAQQINI option is used.

The input string may contain an XML declaration that identifies the encoding of the characters in the XML document. The encoding in the XML declaration must match the encoding of the *string-expression*.

Examples

Example 1: Insert an XML document into the EMP table and preserve the whitespace in the original XML document.

```
INSERT INTO EMP (ID, XVALUE) VALUES(1001,
   XMLPARSE(DOCUMENT '<a xml:space=''preserve''> <b> <c>c</c>b </b> </a>'
            PRESERVE WHITESPACE))
```

XMLPARSE will treat the value for the insert statement as equivalent to the following value:

```
<a xml:space='preserve'> <b> <c>c</c>b </b> </a>
```

Example 2: Insert an XML document into the EMP table and strip the whitespace in the original XML document.

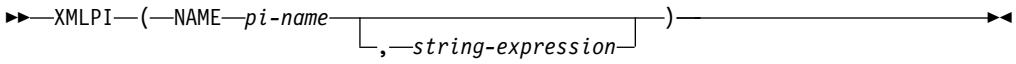
```
INSERT INTO EMP (ID, XVALUE) VALUES(1001,
   XMLPARSE(DOCUMENT
             '<a xml:space=''preserve''> <b xml:space=''default''> <c>c</c>b </b> </a>'
            STRIP WHITESPACE))
```

XMLPARSE will treat the value for the insert statement as equivalent to the following value:

```
<a xml:space='preserve'>
<b xml:space='default'><c>c</c>b </b>
</a>
```

XMLPI

The XMLPI function returns an XML value with a single processing instruction.



NAME *pi-name*
Specifies the name of a processing instruction. The name is an SQL identifier that must be in the form of an XML NCName. See the W3C XML namespace specifications for more details on valid names. The name must not be "xml" in any case combination.

string-expression
An expression that returns a value that is a built-in character or graphic string. It cannot be CHAR or VARCHAR bit data. The resulting string must conform to the content of an XML processing instruction as specified by the following rules:

- The string must not contain the substring '?>' since this substring terminates a processing instruction
- Each character of the string can be any Unicode character, excluding the surrogate blocks, X'FFFE', and X'FFFF'.[77]

If *string-expression* does not conform to the previous rules, an error is returned.

The resulting string becomes the content of the processing instruction.

The result of the function is XML. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value. If *string-expression* is an empty string or is not specified, the result is an empty processing instruction.

Example

- Generate an XML processing instruction.

```
SELECT XMLPI(
     NAME "Instruction", 'Push the red button')
  FROM SYSIBM.SYSDUMMY1
```

This query produces the following result:

```
<?Instruction Push the red button?>
```

- Generate an empty XML processing instruction.

```
SELECT XMLPI(NAME "Warning")
  FROM SYSIBM.SYSDUMMY1
```

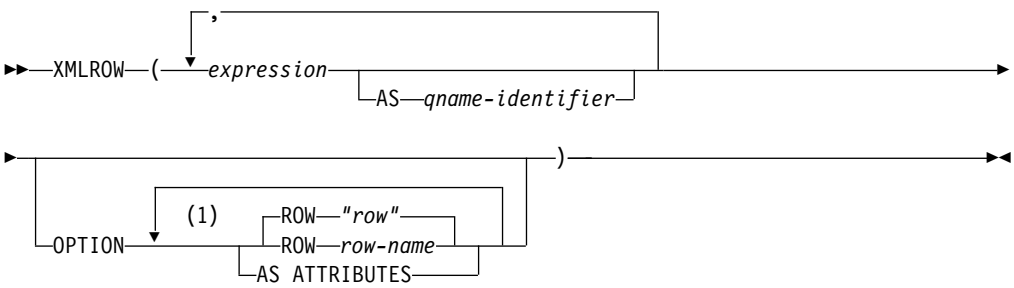
This query produces the following result:

```
<?Warning ?>
```

77. Valid Unicode characters consist of the following Unicode code points: #x9, #xA, #xD, #x20-#xD7FF, #xE000-#xFFFD, #x10000-#x10FFFF.

XMLROW

The XMLROW function returns an XML value that is a well-formed XML document.



Notes:

1 The same clause must not be specified more than once.

expression
The content of each XML element is specified by an expression. The data type of *expression* must not be ROWID or DATALINK or a distinct type that is based on ROWID or DATALINK. When AS ATTRIBUTES is specified, the data type of *expression* must not be XML or a distinct type that is based on XML. The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified.

AS *qname-identifier*
Specifies the XML element name or attribute name as an SQL identifier. The *qname-identifier* must be of the form of an XML qualified name, or QName. See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope. If *qname-identifier* is not specified, *expression* must be a column name. The element name or attribute name is created from the column name using the fully escaped mapping from a column name to a QName.

OPTION
Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

ROW *row-name*
Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

AS ATTRIBUTES
Specifies that each expression is mapped to an attribute value with column name or *qname-identifier* serving as the attribute name. AS ATTRIBUTES cannot be specified if any expression has a result data type of XML.

The result is an XML sequence containing the concatenation of the non-null input XML values.

The result of the function is an XML value. Null values in the input are ignored. If the result of any *expression* can be null, the result can be null; if the result of every *expression* is null, the result is the null value.

Notes

By default, each row in the result set is mapped to an XML value as follows:

- Each row is transformed into an XML element named "row" and each *expression* is transformed into a nested element with the column name or *qname-identifier* as the element name.
- The null handling behavior is NULL ON NULL. A null value for an expression maps to the absence of the subelement. If all expression values are null, the function returns a null value.
- The binary encoding scheme for binary and FOR BIT DATA data types is base64Binary encoding.

Examples

Assume the following table T1 with columns C1 and C2:

```
 C1    C2
----  ----
  1     2
  -     2
  1     -
  -     -
```

- The following example shows an XMLROW query and output fragment with default behavior, using a sequence of row elements to represent the table:

```
  SELECT XMLROW(C1, C2) FROM T1
```

```
<row><C1>1</C1><C2>2</C2></row>
<row><C2>2</C2></row>
<row><C1>1</C1></row>
-
```

- The following example shows an XMLROW query and output fragment with attribute centric mapping. Instead of appearing as nested elements, data is mapped to element attributes:

```
  SELECT XMLROW(C1, C2 OPTION AS ATTRIBUTES) FROM T1
```

```
<row C1="1" C2="2"/>
<row C2="2"/>
<row C1="1"/>
-
```

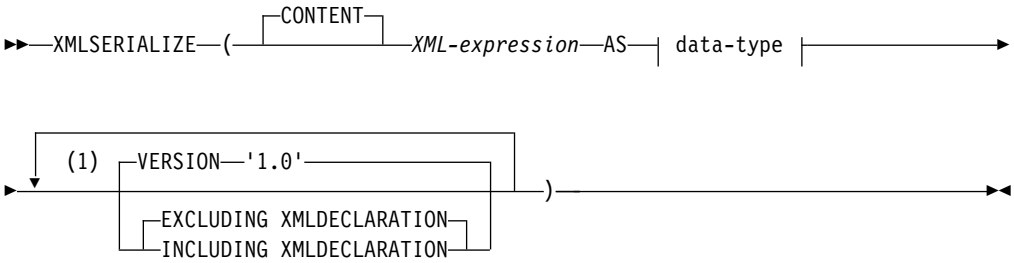
- The following example shows an XMLROW query and output fragment with the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the total of C1 and C2 is returned inside a <total> element:

```
  SELECT XMLROW(C1 AS "column1", C2 AS "column2",
                C1+C2 AS "total" OPTION ROW "entry") FROM T1
```

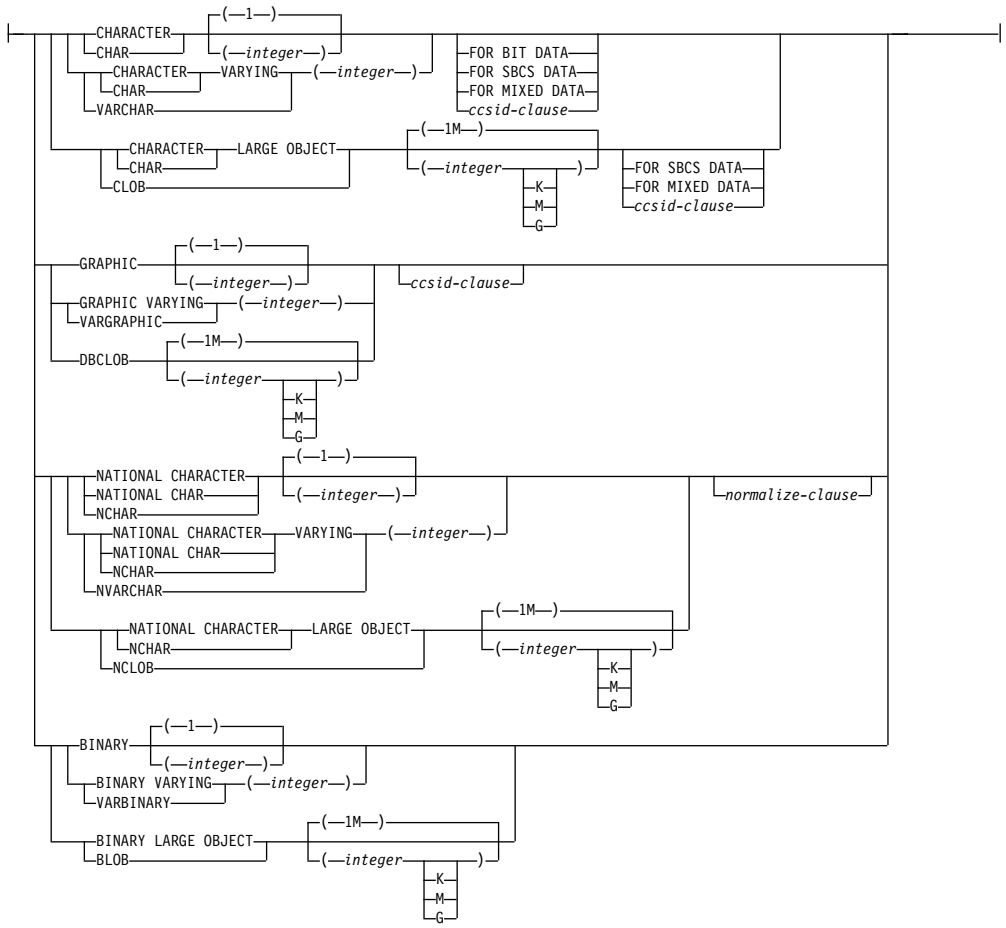
```
<entry><column1>1</column1><column2>2</column2><total>3</total></entry>
<entry><column2>2</column2></entry>
<entry><column1>1</column1></entry>
-
```

XMLSERIALIZE

The XMLSERIALIZE function returns a serialized XML value of the specified data type generated from the *XML-expression* argument.



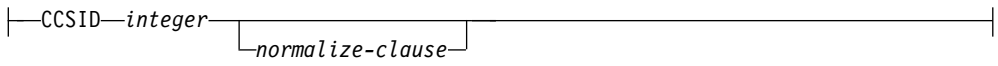
data-type:



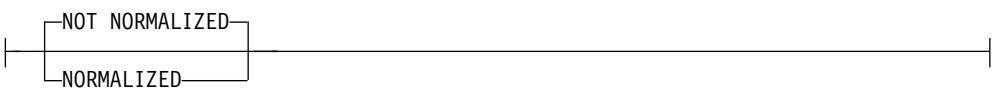
Notes:

1 The same clause must not be specified more than once.

ccsid-clause:



normalize-clause:



CONTENT
Specifies that any XML value can be specified and the result of the serialization is based on this input value.

XML-expression
An expression that returns a value that is a built-in XML string. This is the input to the serialization process.

AS *data-type*
Specifies the result type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the serialized output.

If a CCSID is specified and the *data-type* is GRAPHIC, VARGRAPHIC, or DBCLOB, the CCSID must be a Unicode CCSID.

If the CCSID attribute is not specified, the CCSID is determined as described in "CAST specification" on page 185.

VERSION '1.0'
Specifies the XML version of the serialized value. The only version supported is '1.0' which must be specified as a string constant.

EXCLUDING XMLDECLARATION or INCLUDING XMLDECLARATION
Specifies whether an XML declaration is included in the result. The default is EXCLUDING XMLDECLARATION.

EXCLUDING XMLDECLARATION
Specifies that an XML declaration is not included in the result.

INCLUDING XMLDECLARATION
Specifies that an XML declaration is included in the result. The XML declaration is the string '<?xml version="1.0" encoding="*encoding-name*"?>, where *encoding-name* matches the CCSID of the result data type.

If the result of *XML-expression* can be null, the result can be null; if the result of *XML-expression* is null, the result is the null value.

Examples

Example 1: Serialize into CLOB of UTF-8, the XML value that is returned by the XMLELEMENT function, which is a simple XML element with "Emp" as the element name and an employee name as the element content:

```
SELECT e.empno, XMLSERIALIZE(XMLELEMENT(NAME "Emp",
                                        e.firstnme || ' ' ||e.lastname)
                             AS CLOB(100) CCSID 1208) AS "result"
    FROM employee e WHERE e.lastname = 'SMITH'
```

The result looks similar to the following results:

```
EMPNO   result
-----   --------------------
000250  <Emp>DANIEL SMITH</Emp>
000300  <Emp>PHILIP SMITH</Emp>
```

Example 2: Serialize into a string of BLOB type, the XML value that is returned by the XMLELEMENT function:

```
SELECT XMLSERIALIZE(XMLELEMENT(NAME "Emp",
                                e.firstnme || ' ' ||e.lastname)
                    AS BLOB(1K)
                    VERSION '1.0') AS "result"
    FROM employee e WHERE e.empno = '000300'
```

The result looks similar to the following results:

```
result
--------------------
<Emp>PHILIP SMITH</Emp>
```

Example 3: Serialize into a string of CLOB type, the XML value that is returned by the XMLELEMENT function. Include the XMLDECLARATION:

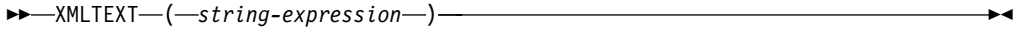
```
SELECT e.empno, e.firstnme, e.lastname,
     XMLSERIALIZE(XMLELEMENT(NAME "xmp:Emp",
                             XMLNAMESPACES('http://www.xmp.com' as "xmp"),
                             XMLATTRIBUTES(e.empno as "serial"),
                             e.firstnme, e.lastname
                             OPTION NULL ON NULL)
                    AS CLOB(1000) CCSID 1208
                    INCLUDING XMLDECLARATION) AS "Result"
    FROM employee e WHERE e.empno = '000300'
```

The result looks similar to the following results:

```
EMPNO   FIRSTNME   LASTNAME   Result
------  ---------  ---------  ----------------------
000300  PHILIP     SMITH      <?xml version="1.0" encoding="UTF-8" ?>
                              <xmp:Emp xmlns:xmp="http://www.xmp.com"
                                  serial="000300">PHILIPSMITH</xmp:Emp>
```

XMLTEXT

The XMLTEXT function returns an XML value that contains the value of *string-expression*.



string-expression
An expression that returns a value of a built-in character or graphic string. It cannot be CHAR or VARCHAR bit data.

The result of the function is an XML value. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value. If the result of *string-expression* is an empty string, the result value is empty text.

Example

- Create a simple XMLTEXT query.

```
VALUES (XMLTEXT (
          'The stock symbol for Johnson&Johnson is JNJ.'))
```

This query produces the following serialized result:

```
The stock symbol for Johnson&amp;Johnson is JNJ.
```

Note that the '&' sign is mapped to '&' when the text is serialized.

- Use XMLTEXT with XMLAGG to construct mixed content. Suppose that the content of table T is as follows:
-

```
SEQNO  PLAINTEXT                                                  EMPHTEXT
-----  ---------------------------------------------------------  ----------
1      This query shows how to construct                          mixed content
2       using XMLAGG and XMLTEXT. Without                         XMLTEXT
3      , XMLAGG will not have text nodes to group with other nodes,  mixed content
       therefore, cannot generate
```

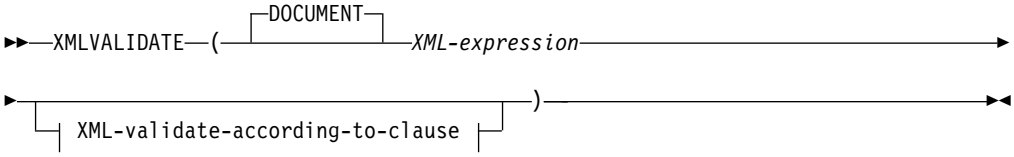
```
SELECT XMLELEMENT(NAME "para",
              XMLAGG(XMLCONCAT(
                                  XMLTEXT(PLAINTEXT),
                                  XMLELEMENT(
                                       NAME "emphasis", EMPHTEXT))
                         ORDER BY SEQNO), '.') AS "result"
  FROM T
```

This query produces the following result:

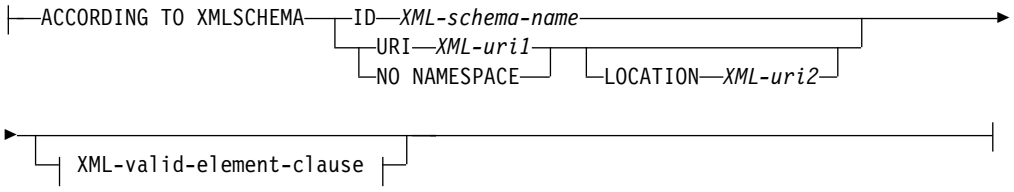
```
result
--------------------------------------------------------------------------------
<para>This query shows how to construct <emphasis>mixed content</emphasis>
using XMLAGG and XMLTEXT. Without <emphasis>XMLTEXT</emphasis>, XMLAGG
will not have text nodes to group with other nodes, therefore, cannot generate
<emphasis>mixed content</emphasis>.</para>
```

XMLVALIDATE

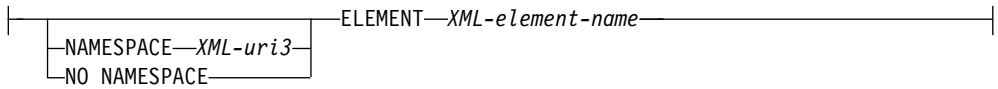
The XMLVALIDATE function returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values and type annotations.



XML-validate-according-to-clause:



XML-valid-element-clause:



DOCUMENT
Specifies that the XML value resulting from *XML-expression* must be a well-formed XML document that conforms to XML Version 1.0.

XML-expression
An expression that returns a value of data type XML. If *XML-expression* is an XML host variable or an implicitly or explicitly typed parameter marker, the function performs a validating parse that strips ignorable whitespace and the CURRENT IMPLICIT XMLPARSE OPTION setting is not considered.

XML-validate-according-to-clause
Specifies the information that is to be used when validating the input XML value.

ACCORDING TO XMLSCHEMA
Indicates that the XML schema information for validation is explicitly specified. If this clause is not included, the XML schema information must be provided in the content of the *XML-expression* value.

ID *XML-schema-name*
Specifies an SQL identifier for the XML schema that is to be used for validation. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned.

URI *XML-uri1*
Specifies the target namespace URI of the XML schema that is to be used for validation. The value of *XML-uri1* specifies a URI as a

character string constant that is not empty. The URI must be the target namespace of a registered XML schema and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema.

NO NAMESPACE
Specifies that the XML schema for validation has no target namespace. The target namespace URI is equivalent to an empty character string that cannot be specified as an explicit target namespace URI.

LOCATION *XML-uri2*
Specifies the XML schema location URI of the XML schema that is to be used for validation. The value of *XML-uri2* specifies a URI as a character string constant that is not empty. The XML schema location URI, combined with the target namespace URI, must identify a registered XML schema, and there must be only one such XML schema registered.

XML-valid-element-clause
Specifies that the XML value in *XML-expression* must have the specified element name as the root element of the XML document.

NAMESPACE *XML-uri3* **or NO NAMESPACE**
Specifies the target namespace for the element that is to be validated. If neither clause is specified, the specified element is assumed to be in the same namespace as the target namespace of the registered XML schema that is to be used for validation.

NAMESPACE *XML-uri3*
Specifies the namespace URI for the element that is to be validated. The value of *XML-uri3* specifies a URI as a character string constant that is not empty. This can be used when the registered XML schema that is to be used for validation has more than one namespace.

NO NAMESPACE
Specifies that the element for validation has no target namespace. The target namespace URI is equivalent to an empty character string which cannot be specified as an explicit target namespace URI.

ELEMENT *xml-element-name*
Specifies the name of a global element in the XML schema that is to be used for validation. The specified element, with implicit or explicit namespace, must match the root element of the value of *XML-expression*.

The result of the function is XML. If the value of *XML-expression* can be null, the result can be null; if the value of *XML-expression* is null, the result is the null value. The CCSID of the result is determined from the *XML-expression*.

The XML validation process is performed on a serialized XML value. Because XMLVALIDATE is invoked with an argument of type XML, this value is automatically serialized prior to validation processing with the following two exceptions:

- If the argument to XMLVALIDATE is an XML host variable or an implicitly or explicitly typed parameter marker, then a validating parse operation is performed on the input value (no implicit non-validating parse is performed and the CURRENT IMPLICIT XMLPARSE OPTION setting is not considered).

- If the argument to XMLVALIDATE is an XMLPARSE invocation using the option PRESERVE WHITESPACE, then the XML parsing and XML validation of the document may be combined into a single validating parse operation.

To validate a document whose root element does not have a namespace, an `xsi:noNamespaceSchemaLocation` attribute must be present on the root element.

Notes

Determining the XML schema: The XML schema can be specified explicitly as part of the XMLVALIDATE invocation or determined from the XML schema information in the input XML value. If the XML schema information is not specified during invocation, the target namespace and the schema location in the input XML value are used to identify the registered schema for validation. If an explicit XML schema is not specified, the input XML value must contain an XML schema information hint. Explicit or implicit XML schema information must identify a registered XML schema, and there must be only one such registered XML schema.

If you do not specify an XML schema document, the database server looks in the input document for an *xsi:schemaLocation* or *xsi:noNamespaceSchemaLocation* attribute that identifies the XML schema document. *xsi:schemaLocation* or *xsi:noNamespaceSchemaLocation* attributes are defined by the XML schema specification and are called XML schema hints. An *xsi:schemaLocation* attribute contains one or more pairs of values that help to locate the XML schema document. The first value in each pair is a namespace and the second value is a hint that indicates where to find the XML schema for the namespace. An *xsi:noNamespaceSchemaLocation* value contains only a hint. If an XML schema document is specified in the XMLVALIDATE function, it overrides the *xsi:schemaLocation* or *xsi:noNamespaceSchemaLocation* attribute.

Specifying the XML schema explicitly on the XMLVALIDATE function avoids the parsing required to locate the XML schema information hint in the XML value.

XML schema authorization: The XML schema used for validation must be registered in the XML schema repository prior to use. The privileges held by the authorization ID of the statement must include at least one of the following:

- USAGE privilege on the XML schema that is to be used during validation
- Administrative authority

Examples

- Validate using the XML schema identified by the XML schema hint in the XML instance document.

```
INSERT INTO T1(XMLCOL)
     VALUES (XMLVALIDATE(?))
```

  Assume that the input parameter marker is bound to an XML value that contains the XML schema information.

```
<po:order
    xmlns:po='http://my.world.com'
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://my.world.com/world.xsd" >
  ...
</po:order>
```

Further, assume that the XML schema that is associated with the target namespace "http://my.world.com" and by schemaLocation hint "http://my.world.com/world.xsd" is found in the XML schema repository

Based on these assumptions, the input XML value will be validated according to that XML schema.

- Validate using the XML schema identified by the SQL name PODOCS.WORLDPO

```
INSERT INTO T1(XMLCOL)
   VALUES(
      XMLVALIDATE(? ACCORDING TO XMLSCHEMA ID PODOCS.WORLDPO))
```

Assuming that the XML schema that is associated with SQL name PODOC.WORLDPO is found in the XML schema repository, the input XML value will be validated and the type annotated according to that XML schema.

- Validate a specified element of the XML value.

```
INSERT INTO T1(XMLCOL)
   VALUES(
      XMLVALIDATE(?
        ACCORDING TO XMLSCHEMA ID FOO.WORLDPO
               NAMESPACE 'http://my.world.com/Mary'
               ELEMENT "po"))
```

Assuming that the XML schema that is associated with SQL name FOO.WORLDPO is found in the XML schema repository, the XML schema will be validated against the element "po", whose namespace is "http://my.world.com/Mary".

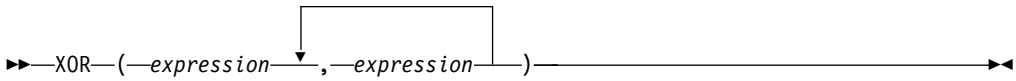
- XML schema is identified by target namespace and schema location.

```
INSERT INTO T1(XMLCOL)
   VALUES(
      XMLVALIDATE(?
        ACCORDING TO XMLSCHEMA URI 'http://my.world.com'
               LOCATION 'http://my.world.com/world.xsd'))
```

Assuming that an XML schema associated with the target namespace "http://my.world.com" and by schemaLocation hint "http://my.world.com/world.xsd" is found in the XML schema repository, the input XML value will be validated according to that XML schema.

XOR

The XOR function returns a string that is the logical XOR of the argument strings. This function takes the first argument string, does an XOR operation with the next string, and then continues to do XOR operations for each successive argument using the previous result. If a character-string argument is shorter than the previous result, it is padded with blanks. If a binary-string argument is shorter than the previous result, it is padded with hexadecimal zeros.



The arguments must be compatible.

expression
An expression that returns a value of any built-in numeric or string data type, but cannot be LOBs. The arguments cannot be mixed data character strings, UTF-8 character strings, or graphic strings. A numeric argument is cast to a character string before evaluating the function. For more information about converting numeric to a character string, see "VARCHAR" on page 531.

The arguments are converted, if necessary, to the attributes of the result. The attributes of the result are determined as follows:

- If all the arguments are fixed-length strings, the result is a fixed-length string of length *n*, where *n* is the length of the longest argument.
- If any argument is a varying-length string, the result is a varying-length string with length attribute *n*, where *n* is the length attribute of the argument with greatest length attribute. The actual length of the result is *m*, where *m* is the actual length of the longest argument.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The CCSID of the result is 65535.

Example

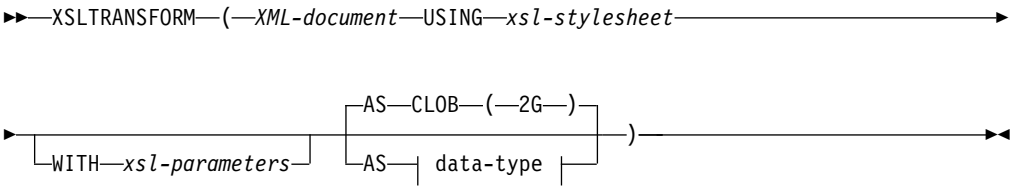
- Assume the host variable L1 is a CHARACTER(2) host variable with a value of X'E1E1', host variable L2 is a CHARACTER(3) host variable with a value of X'F0F000', and host variable L3 is a CHARACTER(4) host variable with a value of X'0000000F'.

```
SELECT XOR(:L1,:L2,:L3)
  FROM SYSIBM.SYSDUMMY1
```

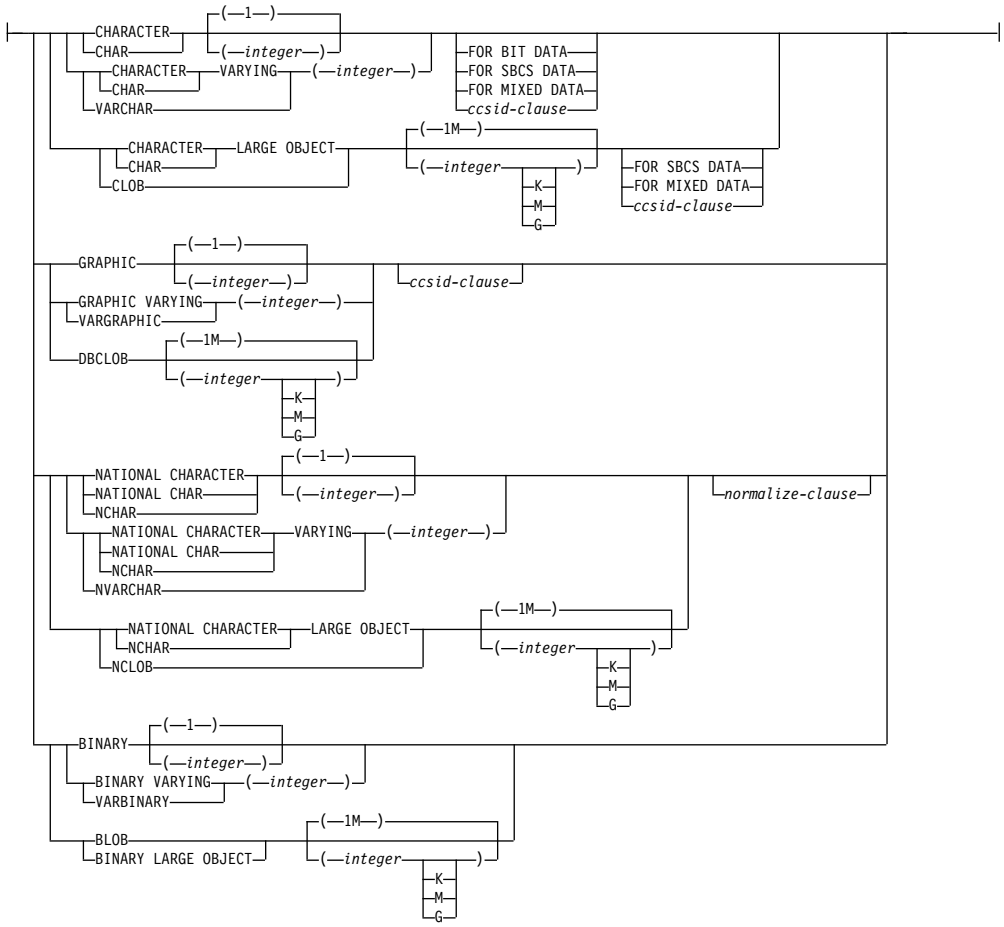
  Returns the value X'1111404F'.

XSLTRANSFORM

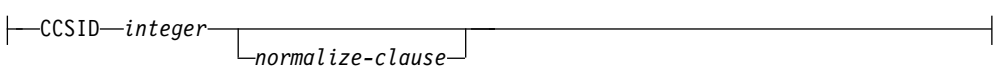
The XSLTRANSFORM transforms an XML document into a different data format. The data can be transformed into any form possible for the XSLT processor, including but not limited to XML, HTML, or plain text.



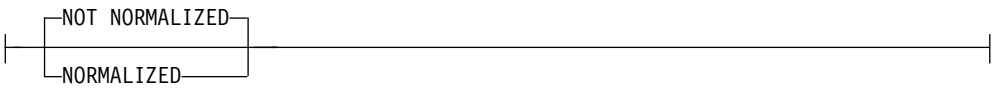
data-type:



ccsid-clause:



normalize-clause:



Use XSLTRANSFORM to convert XML data into other formats including the conversion of XML documents that conform to one XML schema into documents that conform to another XML schema.

XML-document
A character string, Unicode graphic string, binary string, or XML expression that returns a well-formed XML document. This is the document that is transformed using the XSL style sheet specified in *xsl-stylesheet*.

xsl-stylesheet
A character string, Unicode graphic string, binary string, or XML expression that returns a well-formed XML document. The document is an XSL style sheet that conforms to the XSLT Version 1.10 Recommendation. Style sheets incorporating the xsl:include declaration are not supported. This stylesheet is applied to transform the value specified in *xml-document*.

xsl-parameters
A character string, Unicode graphic string, binary string, or XML expression that returns a well-formed XML document. This is a document that provides parameter values to the XSL stylesheet specified in *xsl-stylesheet*. The value of the parameter can be specified as an attribute or as text.

The syntax of the parameter document is as follows:

```
<params xmlns="http://www.ibm.com/XSLTransformParameters">
<param name="..." value="..."/>
<param name="...">enter value here</param> ... </params>
```

Note: The stylesheet document must have `xsl:param` element(s) in it with name attribute values that match the ones specified in the parameter document.

AS *data-type*
Specifies the result data type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the transformed output. The default result data type is CLOB(2G).

If a CCSID is specified and the *data-type* is GRAPHIC, VARGRAPHIC, or DBCLOB, the CCSID must be a Unicode CCSID.

If the CCSID attribute is not specified, the CCSID is determined as if the *XML-document* was cast to *data-type* as described in "CAST specification" on page 185.

The result of the function has the data type specified. CCSID conversion that results in data loss can occur when storing any of the above documents in a character data type.

If either *XML-document* or *xsl-stylesheet* is null, the result is the null value.

Note

Prerequisites: In order to use the XSLTRANSFORM function, the XML Toolkit for IBM i and International Components for Unicode (ICU option) must be installed.

Example

This example illustrates how to use XSLT as a formatting engine. To get set up, first insert the two example documents below into the database.

```
INSERT INTO XML_TAB VALUES
  (1, '<?xml version="1.0"?>
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation = "/home/steffen/xsd/xslt.xsd">
<student studentID="1" firstName="Steffen" lastName="Siegmund"
   age=23 university="Rostock"/>
</students>',

 <?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="headline"/>
<xsl:param name="showUniversity"/>

  <xsl:template match="students">
    <html>
      <head/>
        <body>
          <h1><xsl:value-of select="$headline"/></h1>
          <table border="1">
            <th>
              <tr>
                <td width="80">StudentID</td>
                <td width="200">First Name</td>
                <td width="200">Last Name</td>
                <td width="50">Age</td>
                <xsl:choose>
                  <xsl:when test="$showUniversity ='true'">
                    <td width="200">University</td>
                  </xsl:when>
                </xsl:choose>
              </tr>
            </th>
          <xsl:apply-templates/>
          </table>
        </body>
    </html>
  </xsl:template>
  <xsl:template match="student">
    <tr>
      <td><xsl:value-of select="@studentID"/></td>
      <td><xsl:value-of select="@firstName"/></td>
      <td><xsl:value-of select="@lastName"/></td>
      <td><xsl:value-of select="@age"/></td>
      <xsl:choose>
        <xsl:when test="$showUniversity = 'true'">
          <td><xsl:value-of select="@university"/></td>
        </xsl:when>
      </xsl:choose>
    </tr>
  </xsl:template>
</xsl:stylesheet>');
```

Next, call the XSLTRANSFORM function to convert the XML data into HTML and display it.

```
SELECT XSLTRANSFORM (XML_DOC USING XSL_DOC AS CLOB(1M)) FROM XML_TAB;
```

The result is this document:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1></h1>
<table border="1">
<th>
<tr>
<td width="80">StudentID</td>
<td width="200">First Name</td>
<td width="200">Last Name</td>
<td width="50">Age</td>
</tr>
</th>
<tr>
<td>1</td>
<td>Steffen</td>
<td>Siegmund</td>
<td>23</td>
</tr>
</table>
</body>
</html>
```

In this example, the output is HTML and the parameters influence only what HTML is produced and what data is brought over to it. As such it illustrates the use of XSLT as a formatting engine for end-user output.

YEAR

The YEAR function returns the year part of a value.

expression
An expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 83.
- If *expression* is a number, it must be a date duration or timestamp duration. For the valid formats of datetime durations, see "Datetime operands and durations" on page 173.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a date or a timestamp or a valid character-string representation of a date or timestamp:

  The result is the year part of the value, which is an integer between 1 and 9999.
- If the argument is a date duration or timestamp duration:

  The result is the year part of the value, which is an integer between -9999 and 9999. A nonzero result has the same sign as the argument.

Examples

- Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.

```
SELECT *
  FROM PROJECT
  WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
```

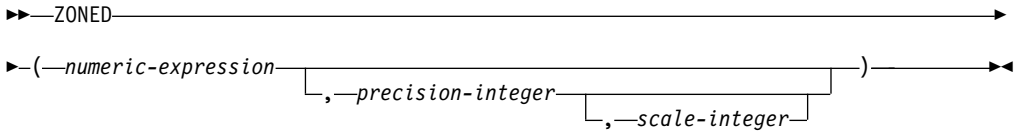
- Select all the projects in the PROJECT table that are scheduled to take less than one year to complete.

```
SELECT *
  FROM PROJECT
  WHERE YEAR(PRENDATE - PRSTDATE) < 1
```

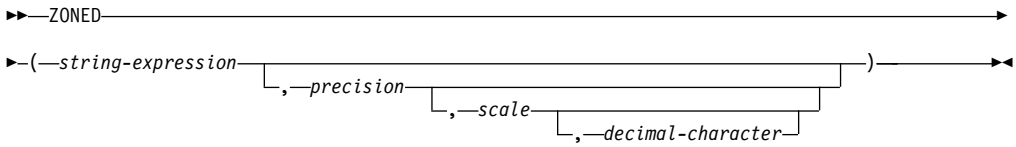
ZONED

The ZONED function returns a zoned decimal representation.

Numeric to Zoned Decimal



String to Zoned Decimal



The ZONED function returns a zoned decimal representation of:

- A number
- A character or graphic string representation of a decimal number
- A character or graphic string representation of an integer
- A character or graphic string representation of a floating-point number
- A character or graphic string representation of a decimal floating-point number

Numeric to Zoned Decimal

numeric-expression
An expression that returns a value of any built-in numeric data type.

precision
An integer constant with a value greater than or equal to 1 and less than or equal to 63.

The default for *precision* depends on the data type of the *numeric-expression*:

- 5 for small integer
- 11 for large integer
- 19 for big integer
- 15 for floating point, decimal, numeric, or nonzero scale binary
- 31 for decimal floating point

scale
An integer constant that is greater than or equal to 0 and less than or equal to *precision*. If not specified, the default is 0.

The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of *precision* and a scale of *scale*. An error is returned if the number of significant decimal digits required to represent the whole part of the number is greater than *precision-scale*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

String to Zoned Decimal

string-expression
An expression that returns a value that is a character-string or graphic-string representation of a number. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant.

precision
An integer constant that is greater than or equal to 1 and less than or equal to 63. If not specified, the default is 15.

scale
An integer constant that is greater than or equal to 0 and less than or equal to *precision*. If not specified, the default is 0.

decimal-character
Specifies the single-byte character constant that was used to delimit the decimal digits in *string-expression* from the whole part of the number. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal separator character. For more information, see "Decimal point" on page 127.

The result is the same number that would result from CAST(*string-expression* AS NUMERIC(*precision*,*scale*)). Digits are truncated from the end if the number of digits to the right of the *decimal-character* is greater than the scale *s*. An error is returned if the number of significant digits to the left of the *decimal-character* (the whole part of the number) in *string-expression* is greater than *precision-scale*. The default decimal separator character is not valid in the substring if the *decimal-character* argument is specified.

The result of the function is a zoned decimal number with precision of *precision* and scale of *scale*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Note

Syntax alternatives: The CAST specification should be used to increase the portability of applications when the precision is specified. For more information, see "CAST specification" on page 185.

Examples

- Assume the host variable Z1 is a decimal host variable with a value of 1.123.

```
SELECT ZONED(:Z1,15,14)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 1.12300000000000.

- Assume the host variable Z1 is a decimal host variable with a value of 1123.

```
SELECT ZONED(:Z1,11,2)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 1123.00.

- Likewise,

```
SELECT ZONED(:Z1,4)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 1123.

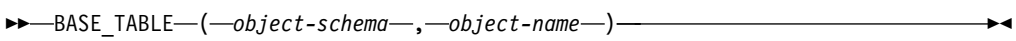
Table functions

Table functions return columns of a table and resemble a table created through a CREATE TABLE statement.

A table function can be used only in the FROM clause of an SQL statement. Table functions can be qualified with a schema name.

BASE_TABLE

The BASE_TABLE function returns the object names and schema names of the object found for an alias.



The schema is SYSPROC.

object-schema
A character or graphic string expression that identifies the SQL or system schema name used to qualify the supplied *object-name*. It cannot be a CLOB or DBCLOB. *object-schema* must have an actual length less than 129 characters. A special value of *LIBL may be specified, in which case, the first instance of a file named *object-name* found in the library list will be used. This name is case sensitive and must not be delimited.

object-name
A character or graphic string expression that identifies the SQL or system name of the object to be resolved. It cannot be a CLOB or DBCLOB. *object-name* must have an actual length less than 129 characters. This name is case sensitive and must not be delimited.

If the specified object does not refer to an alias or it is not found, the result of the function is the input object name and schema.

The result of the function is a table containing a single row with the format shown in the following table. All the columns are nullable.

Table 55. Format of the resulting table for BASE_TABLE

Column name	Data type	Contains
BASESCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table or view referenced by the alias. This is *object-schema* if no alias was found. The name is undelimited and case sensitive.
BASENAME	VARCHAR(128)	Name of the table or view referenced by the alias. This is the *object-name* if no alias was found. The name is undelimited and case sensitive.
SYSTEM_TABLE_SCHEMA	CHAR(10)	System schema name. This column will contain the NULL value if no alias was found or if the alias references a remote RDB. The name may be delimited and is case sensitive.
SYSTEM_TABLE_NAME	CHAR(10)	System table name. This column will contain the NULL value if no alias was found or if the alias references a remote RDB. The name may be delimited and is case sensitive.
MEMBER_NAME	CHAR(10)	The member name that was identified for a member alias. This column will contain the NULL value if no alias was found or if the alias does not reference a specific member. The name may be delimited and is case sensitive.
RDBNAME	VARCHAR(128)	The RDB if the object is a three-part alias for a remote object. This column will contain the NULL value if no alias was found or there is no RDB is associated with the alias.

The CCSID of the result columns is the default CCSID at the current server.

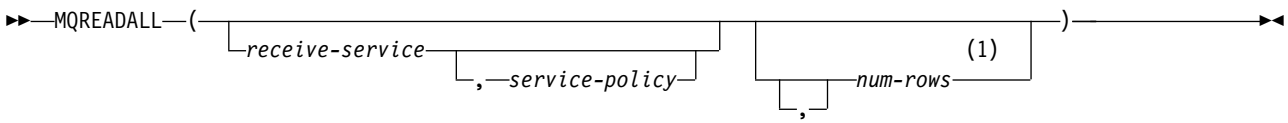
Example

- The following query will return the base table information for every alias identified in SYSTABLES:

```
SELECT C.BASESCHEMA, C.BASENAME
FROM QSYS2.SYSTABLES A,
LATERAL (
    SELECT * FROM TABLE(SYSPROC.BASE_TABLE(A.TABLE_SCHEMA,A.TABLE_NAME)) AS X)
         AS C
WHERE A.TABLE_TYPE='A'
```

MQREADALL

The MQREADALL function returns a table that contains the messages and message metadata from a specified MQSeries location with a VARCHAR column without removing the messages from the queue.



Notes:

1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The MQREADALL function returns a table containing the messages and message metadata from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the messages from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *receive-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

num-rows
An expression that returns a value that is a SMALLINT or INTEGER data type whose value is a positive integer or zero. The value of the expression specifies the maximum number of messages to return.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns are nullable.

Table 56. Format of the resulting table for MQREADALL

Column name	Data type	Contains
MSG	VARCHAR(32000)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	Reserved
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	VARCHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result columns, except for CORRELID and MSGID, is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example receives all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). The messages and all the metadata are returned as a table.

```
SELECT *
 FROM TABLE (MQREADALL ()) AS T
```

- This example receives all the messages from the head of the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY). Only the MSG and CORRELID columns are returned.

```
SELECT T.MSG, T.CORRELID
 FROM TABLE (MQREADALL ('MYSERVICE')) AS T
```

- This example reads the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). Only messages with a CORRELID of '1234' are returned. All columns are returned.

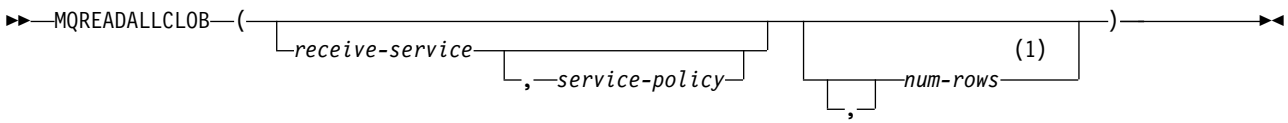
```
SELECT *
 FROM TABLE (MQREADALL ()) AS T
 WHERE T.CORRELID = '1234'
```

- This example receives the first 10 messages from the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). All columns are returned.

```
SELECT *
 FROM TABLE (MQREADALL (10)) AS T
```

MQREADALLCLOB

The MQREADALLCLOB function returns a table that contains the messages and message metadata from a specified MQSeries location with a CLOB column without removing the messages from the queue.



Notes:

1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The MQREADALLCLOB function returns a table containing the messages and message metadata from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the messages from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *receive-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

num-rows
An expression that returns a value that is a SMALLINT or INTEGER data type whose value is a positive integer or zero. The value of the expression specifies the maximum number of messages to return.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns are nullable.

Table 57. Format of the resulting table for MQREADALLCLOB

Column name	Data type	Contains
MSG	CLOB(2M)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	Reserved
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	VARCHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result columns, except for CORRELID and MSGID, is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example receives all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). The messages and all the metadata are returned as a table.

```
SELECT *
 FROM TABLE (MQREADALLCLOB ()) AS T
```

- This example receives all the messages from the head of the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY). Only the MSG and CORRELID columns are returned.

```
SELECT T.MSG, T.CORRELID
 FROM TABLE (MQREADALLCLOB ('MYSERVICE')) AS T
```

- This example reads the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). Only messages with a CORRELID of '1234' are returned. All columns are returned.

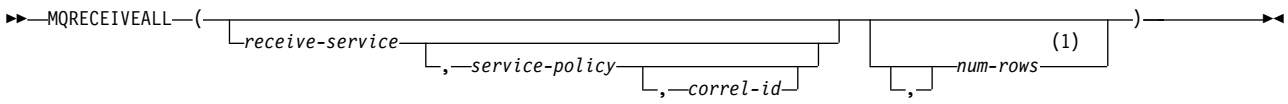
```
SELECT *
 FROM TABLE (MQREADALLCLOB ()) AS T
 WHERE T.CORRELID = '1234'
```

- This example receives the first 10 messages from the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). All columns are returned.

```
SELECT *
 FROM TABLE (MQREADALLCLOB (10)) AS T
```

MQRECEIVEALL

The MQRECEIVEALL function returns a table that contains the messages and message metadata from a specified MQSeries location with a VARCHAR column with removal of the messages from the queue.



Notes:

1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The MQRECEIVEALL function returns a table containing the messages and message metadata from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the messages from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *receive-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. The first message with a matching correlation identifier is returned. For more information about MQSeries Application Messaging, see SQL Programming.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the identical *correl-id* must be specified to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEALL does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified or is an empty string or the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

num-rows
An expression that returns a value that is a SMALLINT or INTEGER data type whose value is a positive integer or zero. The value of the expression specifies the maximum number of messages to return.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns are nullable.

Table 58. Format of the resulting table for MQRECEIVEALL

Column name	Data type	Contains
MSG	VARCHAR(32000)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	Reserved
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	VARCHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result columns, except for CORRELID and MSGID, is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example receives all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). The messages and all the metadata are returned as a table.

```
SELECT *
 FROM TABLE (MQRECEIVEALL ()) AS T
```

- This example receives all the messages from the head of the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY). Only the MSG and CORRELID columns are returned.

```
SELECT T.MSG, T.CORRELID
 FROM TABLE (MQRECEIVEALL ('MYSERVICE')) AS T
```

- This example receives all of the message from the head of the queue specified by the service "MYSERVICE", using the policy "MYPOLICY". Only messages with a CORRELID of '1234' are returned. Only the MSG and CORRELID columns are returned.

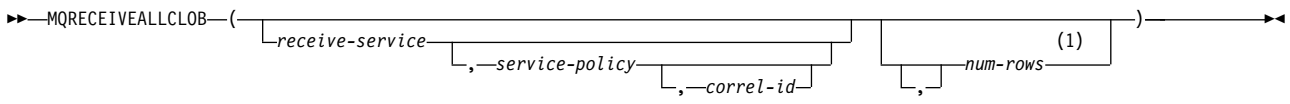
```
SELECT *
 FROM TABLE (MQRECEIVEALL ('MYSERVICE','MYPOLICY','1234')) AS T
```

- This example receives the first 10 messages from the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). All columns are returned.

```
SELECT *
 FROM TABLE (MQRECEIVEALL (10)) AS T
```

MQRECEIVEALLCLOB

The MQRECEIVEALLCLOB function returns a table that contains the messages and message metadata from a specified MQSeries location with a CLOB column with removal of the messages from the queue.



Notes:

1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The MQRECEIVEALLCLOB function returns a table containing the messages and message metadata from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the messages from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the SYSIBM.MQSERVICE table. A service point is a logical end-point from where a message is sent or received. Service point definitions include the name of the MQSeries queue manager and queue. For more information about MQSeries Application Messaging, see SQL Programming.

If *receive-service* is not specified or the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the SYSIBM.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. For more information about MQSeries Application Messaging, see SQL Programming.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. The first message with a matching correlation identifier is returned. For more information about MQSeries Application Messaging, see SQL Programming.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the identical *correl-id* must be specified to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEALLCLOB does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified or is an empty string or the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

num-rows
An expression that returns a value that is a SMALLINT or INTEGER data type whose value is a positive integer or zero. The value of the expression specifies the maximum number of messages to return.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns are nullable.

Table 59. Format of the resulting table for MQRECEIVEALLCLOB

Column name	Data type	Contains
MSG	CLOB(2M)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	Reserved
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	VARCHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result columns, except for CORRELID and MSGID, is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries for IBM i must be installed, configured, and operational.

Example

- This example receives all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). The messages and all the metadata are returned as a table.

```
SELECT *
 FROM TABLE (MQRECEIVEALLCLOB ()) AS T
```

- This example receives all the messages from the head of the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY). Only the MSG and CORRELID columns are returned.

```
SELECT T.MSG, T.CORRELID
 FROM TABLE (MQRECEIVEALLCLOB ('MYSERVICE')) AS T
```

- This example receives all of the message from the head of the queue specified by the service "MYSERVICE", using the policy "MYPOLICY". Only messages with a CORRELID of '1234' are returned. Only the MSG and CORRELID columns are returned.

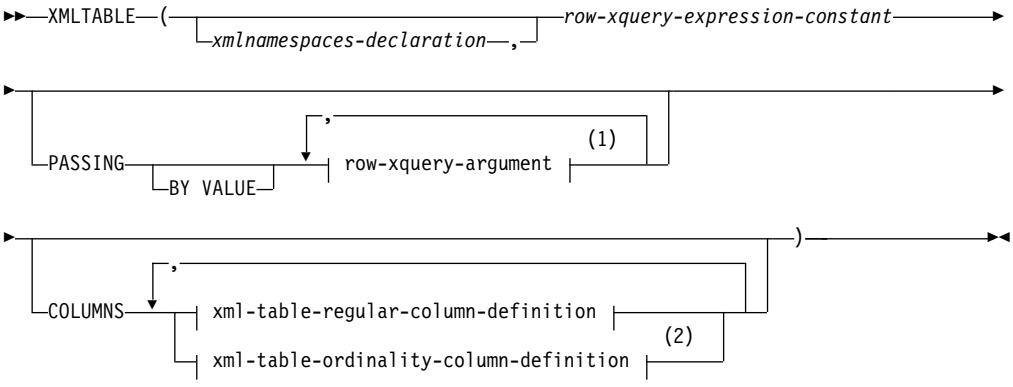
```
SELECT *
 FROM TABLE (MQRECEIVEALLCLOB ('MYSERVICE','MYPOLICY','1234')) AS T
```

- This example receives the first 10 messages from the head of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). All columns are returned.

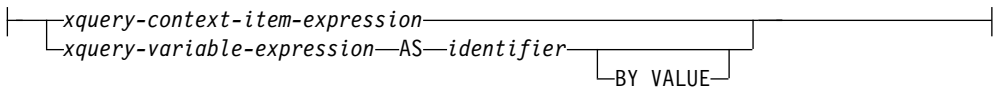
```
SELECT *
 FROM TABLE (MQRECEIVEALLCLOB (10)) AS T
```

XMLTABLE

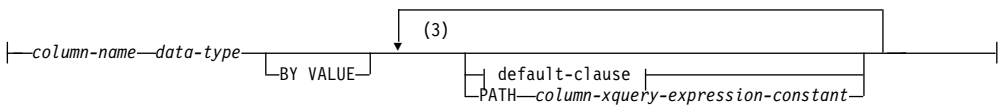
The XMLTABLE function returns a result table from the evaluation of XPath expressions, possibly using specified input arguments as XPath variables. Each item in the result sequence of the row XPath expression represents a row of the result table.



row-xquery-argument:



xml-table-regular-column-definition:



xml-table-ordinality-column-definition:

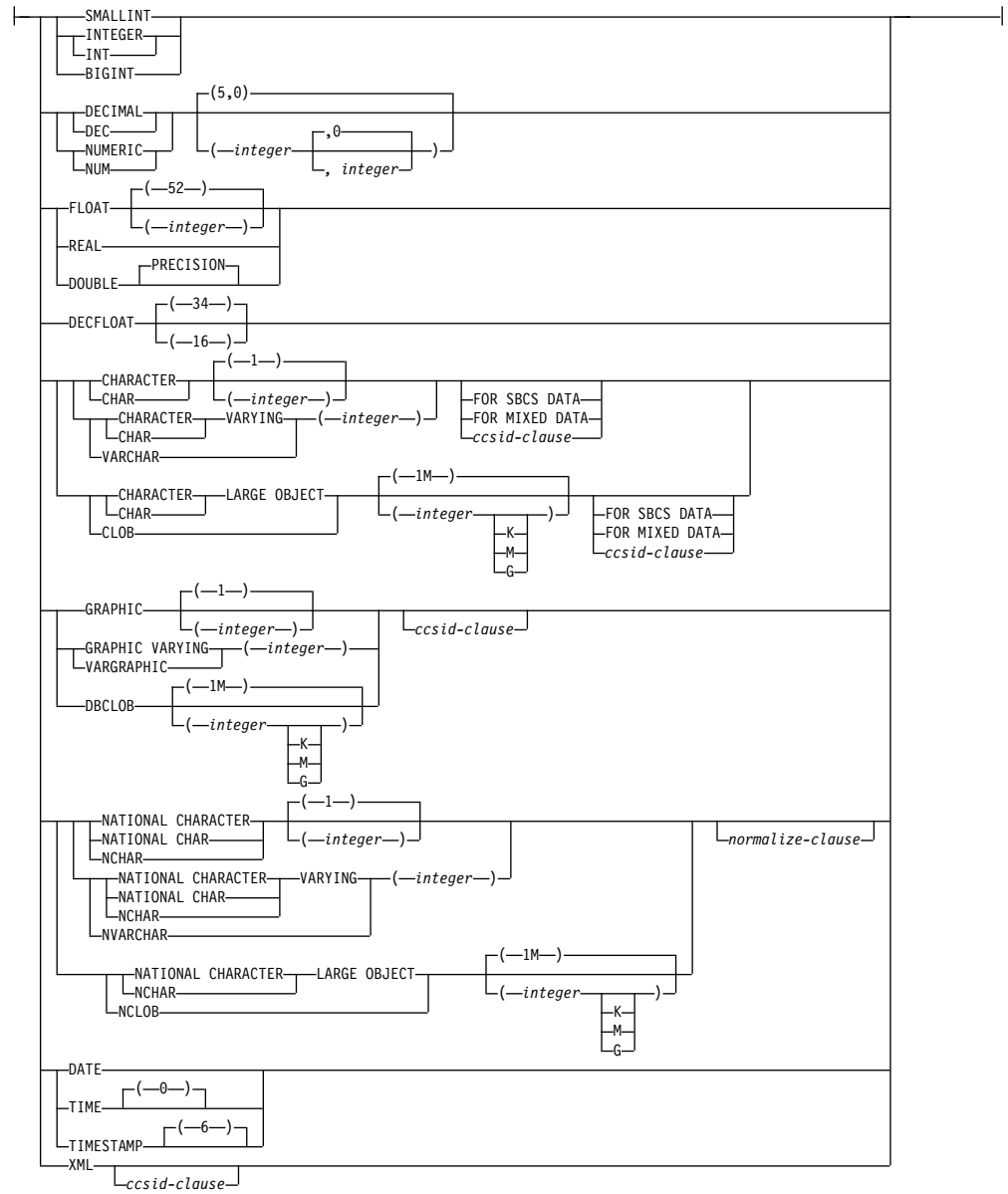
Notes:

1 *xquery-context-item-expression* must not be specified more than one time.

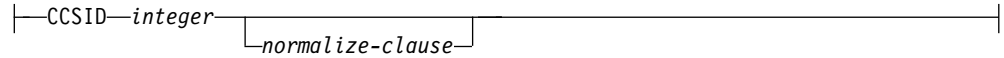
2 The *xml-table-ordinality-column-definition* clause must not be specified more than one time.

3 Neither the *default-clause* nor the PATH clause can be specified more than one time.

data-type:

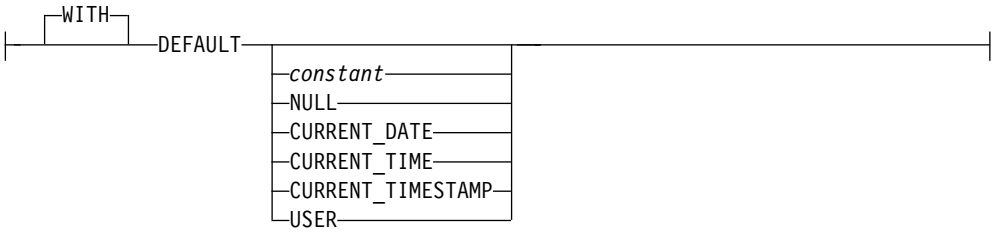


ccsid-clause:



normalize-clause:

default-clause:



The function name cannot be specified as a qualified name.

xmlnamespaces-declaration
Specifies one or more XML namespace declarations, using the XMLNAMESPACES function, that become part of the static context of the *row-xquery-expression-constant* and the *column-xquery-expression-constant*. The set of statically known namespaces for XPath expressions which are arguments of XMLTABLE is the combination of the pre-established set of statically known namespaces and the namespace declarations specified in this clause. The XPath prolog within an XPath expression can override these namespaces.

If *xmlnamespaces-declaration* is not specified, only the pre-established set of statically known namespaces apply to the XPath expressions.

row-xquery-expression-constant
Specifies an SQL string constant that is interpreted as an XPath expression using supported XPath language syntax. This expression determines the number of rows in the result table. The expression is evaluated using the optional set of input XML values that is specified in *row-xquery-argument*, and returns an output XPath sequence where one row is generated for each item in the sequence. If the sequence is empty, the result of XMLTABLE is an empty table. *row-xquery-expression-constant* must not be an empty string or a string of all blanks.

PASSING
Specifies input values and the manner in which these values are passed to the XPath expression specified by *row-xquery-expression-constant*.

BY VALUE
Specifies that any XML arguments are passed by value. When XML values are passed by value, the XPath evaluation uses a copy of the XML data. This is the default behavior. DB2 for i binds the XPath variable expression to a document node that represents the XML input value.

This clause has no impact on how non-XML values are passed. Non-XML values always create a copy of the value during the cast to XML.

row-xquery-argument
Specifies an argument that is to be passed to the XPath expression specified by *row-xquery-expression-constant*. *row-xquery-argument* specifies an SQL expression that is evaluated before being passed to the XPath expression.

If the data type of *row-xquery-argument* is not XML, the result of the expression is converted to XML. For *xquery-variable-expression*, a null value is converted to an XML empty sequence.

How *row-xquery-argument* is used in the XPath expression depends on whether the argument is specified as an *xquery-context-item-expression* or an *xquery-variable-expression*.

row-xquery-argument must not contain a NEXT VALUE or PREVIOUS VALUE expression or an OLAP specification.

xquery-context-item-expression
Specifies an SQL expression that returns a value that is XML or that is a type that has a supported conversion to XML.

xquery-context-item-expression specifies the initial context item for the *row-xquery-expression*. The value of the initial context item is the result of *xquery-context-item-expression* after being converted to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-variable-expression
Specifies an SQL expression whose value is available to the XPath expression specified by *row-xquery-expression-constant* during execution. The expression must return a value that is XML or that is a type that has a supported conversion to XML.

xquery-variable-expression specifies an argument that will be passed to *row-xquery-expression-constant* as an XPath variable. If *xquery-variable-expression* is the null value, the XPath variable is set to an XML empty sequence. The scope of the XPath variables that are created from the PASSING clause is the XPath expression specified by *row-xquery-expression-constant*.

AS *identifier*
Specifies that the value generated by *xquery-variable-expression* will be passed to *row-xquery-expression-constant* as an XPath variable. The *identifier* is a name that must be in the form of an XML NCName. See the W3C XML namespace specifications for more details on valid names. The leading dollar sign ($) that precedes variable names in the XPath language must not be included as part of *identifier*. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier.

BY VALUE
Specifies that *xquery-variable-expression* is passed by value. When XML values are passed by value, the XPath evaluation uses a copy of the XML data. DB2 for i binds the XPath variable expression to a document node that represents the XML input value. If BY VALUE is not specified following *xquery-variable-expression*, XML arguments are passed using the default passing mechanism that is provided through the syntax that follows the PASSING keyword.

This clause is only valid for input values with the XML data type. Non-XML values always create a copy of the value during the cast to XML.

Table 60. Supported SQL to XML conversions

SQL type	XML type	Notes
SMALLINT	xs:integer	
INTEGER	xs:integer	
BIGINT	xs:integer	
DECIMAL NUMERIC	xs:decimal	Decimal numbers with a precision greater than 34 can lose precision during processing.
FLOAT DOUBLE DECFLOAT	xs:double	

Table 60. Supported SQL to XML conversions (continued)

SQL type	XML type	Notes
CHAR VARCHAR CLOB GRAPHIC VARGRAPHIC DBCLOB	xs:string	When character string values are cast to XML values, the resulting xs:string atomic value cannot contain illegal XML characters. If the input character string is not in Unicode, the input characters are converted to Unicode. CHAR and VARCHAR strings cannot have a CCSID of 65535 or be defined for bit data.
DATE	xs:date	The xs:date value will not have a timezone component. For comparisons, the timezone is implicitly assumed to be UTC. If needed, the fn:adjust-timezone() function can be used to explicitly set the timezone.
TIME	xs:time	The xs:time value will not have a timezone component. For comparisons, the timezone is implicitly assumed to be UTC. If needed, the fn:adjust-timezone() function can be used to explicitly set the timezone.
TIMESTAMP	xs:dateTime	The xs:dateTime value will not have a timezone component. For comparisons, the timezone is implicitly assumed to be UTC. If needed, the fn:adjust-timezone() function can be used to explicitly set the timezone.

COLUMNS

Specifies the output columns of the result table including the column name, data type, and how the column value is computed for each row. If this clause is not specified, a single unnamed column of type XML is returned with the value based on the sequence item from evaluating the XPath expression in the *row-xquery-expression* (equivalent to specifying `PATH '.'`). To reference this result column, a *column-name* must be specified in the *correlation-clause* following the table function.

The sum of all the result column lengths cannot exceed 64K bytes. For information on the byte counts of columns according to data type, see "Maximum row sizes" on page 1027. Assume the number of *row-xquery-arguments* is N. There must be no more than (249-(N*2))/2 columns.

xml-table-regular-column-definition

Specifies one output column of the result table including the column name, data type, and an XPath expression to extract the value from the sequence item for the row.

column-name

Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the result table.

data-type

Specifies the data type of the column. For CHAR and VARCHAR columns, the CCSID cannot be 65535.

BY VALUE

Specifies that the result column is returned by value. When XML

values are returned by value, a copy of the XML data is returned. This is the default behavior. DB2 for i constructs a document node for the XML result when the value is returned from the table function. This clause must not be specified for a column with a data type that is not XML.

default-clause
Specifies a default value for the column. For XMLTABLE result columns, the default is applied when the processing of the XPath expression contained in *column-xquery-expression-constant* returns an empty sequence.

PATH *column-xquery-expression-constant*
Specifies a string constant that is interpreted as an XPath expression using supported XPath language syntax. The *column-xquery-expression-constant* specifies an XPath expression that determines the column value with respect to an item that is the result of evaluating the XPath expression in *row-xquery-expression-constant*. Given an item from the result of processing the *row-query-expression-constant* as the externally provided context item, the *column-xquery-expression-constant* is evaluated and returns an output sequence. The column value is determined based on this output sequence as follows:

- If an empty sequence is returned, the *default-clause* provides the value of the column.
- If an empty sequence is returned and no *default-clause* was specified, the null value is assigned to the column.
- If a non-empty sequence is returned, the value is converted to the data-type specified for the column. An error could be returned from processing this implicit conversion.

The value for *column-xquery-expression-constant* must not be an empty string or a string of all blanks. If this clause is not specified, the default XPath expression is the *column-name*.

xml-table-ordinality-column-definition
Specifies the ordinality column of the result table.

colum-name
Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the result table.

FOR ORDINALITY
Specifies that *column-name* is the ordinality column of the result table. The data type of this column is BIGINT. The value of this column in the result table is the sequential number of the item for the row in the resulting sequence from evaluating the XPath expression in *row-xquery-expression-constant*.

Table 61. Supported XML to SQL result column conversions

XML type	SQL type	Notes
xs:integer	SMALLINT INTEGER BIGINT	

Table 61. Supported XML to SQL result column conversions (continued)

XML type	SQL type	Notes
xs:decimal	DECIMAL NUMERIC	The resulting xs:decimal value is converted, if necessary, to the precision and scale of the target data type. The necessary number of leading zeros is added or removed. In the fractional part of the number, the necessary number of trailing zeros is added or the necessary number of digits is eliminated. This truncation behavior is similar to the behavior of the cast from DECIMAL to DECIMAL. Decimal numbers with a precision greater than 34 can lose precision during processing.
xs:double	FLOAT DOUBLE REAL DECFLOAT	If the target type is FLOAT, DOUBLE, or REAL and the source XML value after the XPath cast is an xs:double value of INF, -INF, or NaN, an error is returned. If the source value is an xs:double negative zero, the value is converted to positive zero. If the source value is beyond the range of the target data type, an overflow error is returned. If the source value contains more significant digits than the precision of the target data type, the source value is rounded to the precision of the target data type. If the target type is DECFLOAT and the source XML value is an xs:double value of INF, -INF, or NaN, the result will be the corresponding special DECFLOAT values INF, -INF, or NaN. If the source value is an xs:double negative zero, the result is negative zero. If the target type is DECFLOAT(16) and the source value is beyond the range of DECFLOAT(16), an overflow error is returned. If the source value has more than 16 significant digits, the value is rounded according to the ROUNDING mode that is in effect. This rounding behavior is the same as what is used during the cast of DECFLOAT(34) to DECFLOAT(16).
xs:string	CHAR VARCHAR CLOB GRAPHIC VARGRAPHIC DBCLOB	The resulting XML value is converted, if necessary, to the CCSID of the target data type using the rules described in “Conversion rules for assignments” on page 104 before it is converted to the target type with a limited length. Truncation occurs if the specified length limit is smaller than the length of the resulting string after CCSID conversion. A warning occurs if any non-blank characters are truncated. If the target type is a fixed-length string type (CHAR or GRAPHIC) and the specified length of the target type is greater than the length of the resulting string from CCSID conversion, blanks are padded at the end. This truncation and padding behavior is similar to retrieval assignment of character or graphic strings.

Table 61. Supported XML to SQL result column conversions (continued)

XML type	SQL type	Notes
xs:date	DATE	The resulting XML value is adjusted to UTC time and the time zone component is removed. The year part of the resulting xs:date value must be in the range of 0001 to 9999.
xs:time	TIME	The resulting XML value is adjusted to UTC time and the time zone component is removed. Any fractional seconds are truncated from the result.
xs:dateTime	TIMESTAMP	The resulting XML value is adjusted to UTC time and the time zone component is removed. The year part of the resulting xs:dateTime value must be in the range of 0001 to 9999. If the resulting xs:dateTime value contains fractional seconds, those digits are truncated so that the result contains six digits of fractional seconds precision

The result of the function is a table. If the evaluation of any of the XPath expressions results in an error, then the XMLTABLE function returns the XPath error.

Example

- List as a table result the purchase order items for orders with a status of 'NEW'.

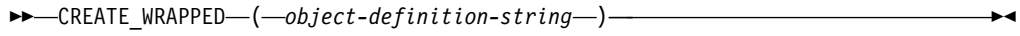
```
SELECT U."PO ID", U."Part #", U."Product Name",
       U."Quantity", U."Price", U."Order Date"
  FROM PURCHASEORDER P,
       XMLTABLE(XMLNAMESPACES('http://podemo.org' AS "pod"),
                '$po/PurchaseOrder/itemlist/item' PASSING P.PORDER AS "po"
                COLUMNS "PO ID"         INTEGER       PATH '../../@POid',
                        "Part #"        CHAR(6)       PATH 'product/@pid',
                        "Product Name"  CHAR(50)      PATH 'product/pod:name',
                        "Quantity"      INTEGER       PATH 'quantity',
                        "Price"         DECIMAL(9,2)  PATH 'product/pod:price',
                        "Order Date"    TIMESTAMP     PATH '../../dateTime'
                ) AS U
  WHERE P.STATUS = 'NEW'
```

Chapter 4. Procedures

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the XML procedures.

CREATE_WRAPPED

The CREATE_WRAPPED procedure transforms a readable DDL statement into an obfuscated DDL statement and then deploys the object in the database.



In an obfuscated DDL statement, the procedural logic and embedded SQL statements are scrambled in such a way that any intellectual property in the logic cannot be easily extracted.

The schema is SYSIBMADM.

object-definition-string
A string of type CLOB containing a DDL statement. It can be one of the following SQL statements (SQLSTATE 5UA0O):

- CREATE FUNCTION (SQL scalar)
- CREATE FUNCTION (SQL table)
- CREATE PROCEDURE (SQL)

The procedure transforms the input into an obfuscated DDL statement string and then dynamically executes that DDL statement. The encoding consists of a prefix of the original statement up to and including the routine signature or trigger name, followed by the keyword WRAPPED. This keyword is followed by information about the application server that invoked the function. The information has the form *pppvvrrm* where:

- *ppp* identifies the product using the following 3 characters:
  - QSQ for DB2 for i
  - SQL for DB2 Database for Linux, UNIX, and Windows
- *vv* is a two-digit version identifier, such as '09'
- *rr* is a two-digit release identifier, such as '07'
- *m* is a one-character modification level identifier, such as '0'

For example DB2 for i version 7.1 is identified as 'QSQ07010'.

This application server information is followed by a string of letters (a-z and A-Z), digits (0-9), underscores, and colons.

The encoded DDL statement may be up to one-third longer than the plain text form of the statement. If the result exceeds the maximum length for SQL statements, an error is issued (SQLSTATE 54001).

Note

The encoding of the statement is meant to obfuscate the content and should not be considered as a form of strong encryption.

Example

Produce an obfuscated version of a function that computes a yearly salary from an hourly wage given a 40 hour work week.

```
CALL CREATE_WRAPPED('CREATE FUNCTION salary(wage DECFLOAT)
                                RETURNS DECFLOAT RETURN wage * 40 * 52');

SELECT ROUTINE_DEFINITION FROM QSYS2.SYSROUTINES
    WHERE routine_name ='SALARY' AND routine_schema = CURRENT SCHEMA;
```

Upon successful completion of the CALL statement, the ROUTINE_DEFINITION column in QSYS2.SYSROUTINES for the row corresponding to routine 'SALARY' would be something of the form:

```
WRAPPED QSQ07010 <encoded-suffix>
```

XDBDECOMPXML

The XDBDECOMPXML procedure extracts values from serialized XML data and populates relational tables with the values.

Authorization

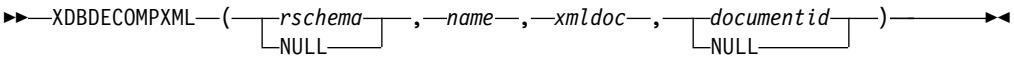
The privileges held by the authorization ID of the statement must include the following:

- The following system authorities:
  - The system authority *EXECUTE on the XDBDECOMPXML service program associated with the procedure, and
  - The system authority *EXECUTE on the SYSPROC library.

The privileges held by the authorization ID of the statement must include:

- The INSERT privilege on any tables specified in the annotations, or
- Administrative authority

Syntax



Description

The schema is SYSPROC.

The XDBDECOMPXML stored procedure uses an XML schema which contains annotations that indicate which columns and tables should be used to store the decomposed XML values. The referenced XML schema must exist in the XSR and must be enabled for decomposition. You can enable an XML schema for decomposition by using the XSR_COMPLETE stored procedure. If your XML schema references tables that did not exist when you invoked the XSR_COMPLETE stored procedure, DB2 will return an error.

rschema
An input parameter of type VARCHAR(128) that specifies the SQL schema for the XML schema. It must be a valid SQL identifier. The SQL schema is one part of the qualified name used to identify this XML schema in the XSR. (The other part of the name is supplied by the *name* parameter). This parameter can have the NULL value which indicates that *name* is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

If *rschema* is specified, it cannot be QSYS, QSYS2, SYSIBM, SYSPROC, or QTEMP.

name
An input parameter of type VARCHAR(128) that specifies the name of the XML schema. It must be a valid SQL identifier. The complete name for the XML schema for which decomposition is to be performed is *rschema.name*. The XML schema name must already exist and be enabled for decomposition as a result of calling the XSR_COMPLETE stored procedure. This parameter cannot have the NULL value.

xmldoc
An input parameter of type BLOB(2G) that points to the XML value that is to be decomposed. This parameter cannot be null.

documentid
An input parameter of type VARCHAR(1024) that contains a string that the caller can use to identify the input XML document. This parameter can be null.

XSR_ADDSCHEMADOC

The XSR_ADDSCHEMADOC stored procedure adds every XML schema other than the primary XML schema document to the XSR.

Authorization

Each XML schema in the XSR can consist on one or more XML schema documents. When an XML schema consists of multiple documents, you need to call XSR_ADDSCHEMADOC for the additional documents.

The privileges held by the authorization ID of the statement must include the following:

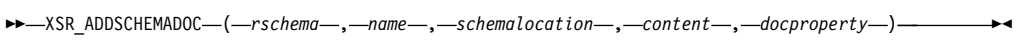
- The following system authorities:
  - The system authority *EXECUTE on the service program associated with the procedure,
  - The system authority *EXECUTE on the SYSPROC library,
  - The system authority *EXECUTE on the library containing the *SQLXSR object, and
  - The ALTER privilege for the *SQLXSR object.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the XSROBJECTCOMPONENTS and XSROBJECTHIERARCHIES catalog tables:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2.
- Administrative authority

The user ID of the caller of the procedure must have the EXECUTE privilege on the XSR_ADDSCHEMADOC stored procedure and must be the definer of the XSR object as recorded in the XSROBJECTS catalog table.

Syntax



Description

The schema is SYSPROC.

rschema
An input parameter of type VARCHAR(128) that specifies the SQL schema for the XML schema. It must be a valid SQL identifier. The SQL schema is one part of the qualified name used to identify this XML schema in the XSR. (The other part of the name is supplied by the *name* parameter). This parameter can have the NULL value which indicates that *name* is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

If *rschema* is specified, it cannot be QSYS, QSYS2, SYSIBM, SYSPROC, or QTEMP.

name
An input parameter of type VARCHAR(128) that specifies the name of the XML schema. It must be a valid SQL identifier. The complete name for the XML schema is *rschema.name*. The XML schema must already exist as a result of calling the XSR_REGISTER stored procedure and XML schema registration cannot yet be completed. This parameter cannot have the NULL value.

schemalocation
An input parameter of type VARCHAR(1000), which can have the NULL value, that indicates the schema location of the primary XML schema document to which the XML schema document is being added. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute. The document that references the *schemalocation* must use a valid URI format.

content
An input parameter of type BLOB(30M) that contains the content of the XML schema document being added. This argument cannot have the NULL value; an XML schema document must be supplied. The content of the XML schema document must be encoded in UTF-8.

docproperty
An input parameter of type BLOB(5M) that indicates the properties for the XML schema document being added. This argument can have the NULL value; otherwise, the value is an XML document.

Example

The following example calls the XSR_ADDSCHEMADOC stored procedure:

```
CALL SYSPROC.XSR_ADDSCHEMADOC(
  'MyLib',
  'MySchema',
  'http://myschema/addschema1.xsd',
  :schema_content,
  :schema_docproperties)
```

XSR_COMPLETE

The XSR_COMPLETE procedure is the final stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XSR. An XML schema is not available for validation until the schema registration completes through a call to this stored procedure.

Authorization

The privileges held by the authorization ID of the statement must include the following:

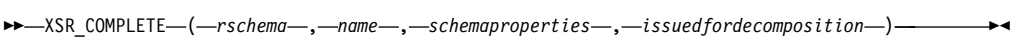
- The following system authorities:
  - The system authority *EXECUTE on the service program associated with the procedure,
  - The system authority *EXECUTE on the SYSPROC library,
  - The system authority *EXECUTE on the library containing the *SQLXSR object, and
  - The ALTER privilege for the *SQLXSR object.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the XSROBJECTS, XSROBJECTCOMPONENTS, XSROBJECTHIERARCHIES, and XSRANNOTATIONINFO catalog tables:
  - The UPDATE privilege on XSROBJECTS, XSROBJECTCOMPONENTS, and XSROBJECTHIERARCHIES,
  - The INSERT privilege on XSRANNOTATIONINFO, and
  - The system authority *EXECUTE on library QSYS2.
- Administrative authority

The user ID of the caller of the procedure must have the EXECUTE privilege on the XSR_COMPLETE stored procedure.

Syntax



Description

The schema is SYSPROC.

rschema
An input parameter of type VARCHAR(128) that specifies the SQL schema for the XML schema. It must be a valid SQL identifier. The SQL schema is one part of the qualified name used to identify this XML schema in the XSR. (The other part of the name is supplied by the *name* parameter). This parameter can have the NULL value which indicates that *name* is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

If *rschema* is specified, it cannot be QSYS, QSYS2, SYSIBM, SYSPROC, or QTEMP.

name
An input parameter of type VARCHAR(128) that specifies the name of the

XML schema. It must be a valid SQL identifier. The complete name for the XML schema for which a completion check is to be performed is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER stored procedure, and XML schema registration cannot yet be completed. This parameter cannot have the NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this parameter.

schemaproperties
An input parameter of type BLOB(5M) that specifies properties, if any, associated with the XML schema. The values for this parameter is either NULL, if there are no associated properties, or an XML document representing the properties for the XML schema.

issuedfordecomposition
An input parameter of type INTEGER that indicates if an XML schema is to be used for decomposition. If an XML schema is to be used for decomposition, this value should be set to 1; otherwise, it should be set to 0.

Example

The following example calls the XSR_COMPLETE stored procedure:

```
CALL SYSPROC.XSR_COMPLETE(
  'MyLib',
  'MySchema',
  :schemaproperty_host_var,
  0)
```

XSR_REGISTER

The XSR_REGISTER procedure is the first stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR).

Authorization

The user that calls this stored procedure is considered the creator of this XML schema. DB2 obtains the namespace attribute from the schema document when XSR_COMPLETE is invoked.

The privileges held by the authorization ID of the statement must include the following:

- The following system authorities:
  - The system authority *EXECUTE on the service program associated with the procedure, and
  - The system authority *EXECUTE on the SYSPROC library.

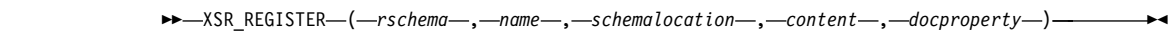
The privileges held by the authorization ID of the statement must include as least one of the following:

- The privilege to create in the schema
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the XSROBJECTS, XSROBJECTCOMPONENTS, and XSROBJECTHIERARCHIES catalog tables:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2.
- Administrative authority

Syntax



Description

The schema is SYSPROC.

rschema
An input parameter of type VARCHAR(128) that specifies the SQL schema for the XML schema. It must be a valid SQL identifier. The SQL schema is one part of the qualified name used to identify this XML schema in the XSR. (The other part of the name is supplied by the *name* parameter). This parameter can have the NULL value which indicates that *name* is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

If *rschema* is specified, it cannot be QSYS, QSYS2, SYSIBM, SYSPROC, or QTEMP.

name
An input parameter of type VARCHAR(128) that specifies the name of the

XML schema. It must be a valid SQL identifier. The complete name for the XML schema is *rschema.name* and should be unique among all objects in the XSR.

schemalocation
An input parameter of type VARCHAR(1000), which can have the NULL value, that indicates the schema location of the primary XML schema document. This parameter is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

content
An input parameter of type BLOB(30M) that contains the content of the primary XML schema document. This parameter cannot have the NULL value; an XML schema document must be supplied. The content of the XML schema document must be encoded in UTF-8.

docproperty
An input parameter of type BLOB(5M) that indicates the properties for the primary XML schema document. This parameter can have the NULL value; otherwise, the value is an XML document.

Example

The following example calls the XSR_REGISTER stored procedure:

```
CALL SYSPROC.XSR_REGISTER(
  'MyLib',
  'MySchema',
  'http://myschema/primary.xsd',
  :content_host_var,
  :docproperty_host_var)
```

XSR_REMOVE

The XSR_REMOVE procedure is used to remove all components of an XML schema. After the XML schema is removed, you can reuse the name of the removed XML schema when you register a new XML schema.

Authorization

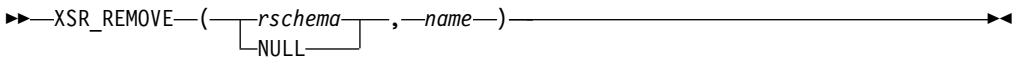
The privileges held by the authorization ID of the statement must include the following:

- The following system authorities:
  - The system authority *EXECUTE on the service program associated with the procedure, and
  - The system authority *EXECUTE on the SYSPROC library.

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority *OBJOPR and *OBJEXIST on the object associated with the XSR object, and
  - The system authority *EXECUTE on the library that contains the XSR object to be dropped, and
  - The DELETE privilege on the XSROBJECTS, XSROBJECTCOMPONENTS, XSROBJECTHIERARCHIES, and XSRANNOTATIONINFO catalog tables, and
  - The system authority *EXECUTE on library QSYS2.
- Administrative authority

Syntax



Description

The schema is SYSPROC.

rschema
An input parameter of type VARCHAR(128) that specifies the SQL schema for the XML schema. It must be a valid SQL identifier. The SQL schema is one part of the qualified name used to identify this XML schema in the XSR. (The other part of the name is supplied by the *name* parameter). This parameter can have the NULL value which indicates that *name* is implicitly qualified based on the rules specified in "Qualification of unqualified object names" on page 63.

If *rschema* is specified, it cannot be QSYS, QSYS2, SYSIBM, SYSPROC, or QTEMP.

name
An input parameter of type VARCHAR(128) that specifies the name of the XML schema. It must be a valid SQL identifier. The complete name for the XML schema that is to be removed is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER stored procedure. This parameter cannot have the NULL value.

Example

The following example calls the XSR_REMOVE stored procedure:

```
CALL SYSPROC.XSR_REMOVE(
  'MyLib',
  'MySchema')
```

Chapter 5. Queries

A *query* specifies a result table or an intermediate result table. A query is a component of certain SQL statements.

The three forms of a query are the *subselect*, the *fullselect*, and the *select-statement*. There is another SQL statement that can be used to retrieve at most a single row described under "SELECT INTO" on page 1345.

Authorization

For any form of a query, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the statement,
  - The SELECT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

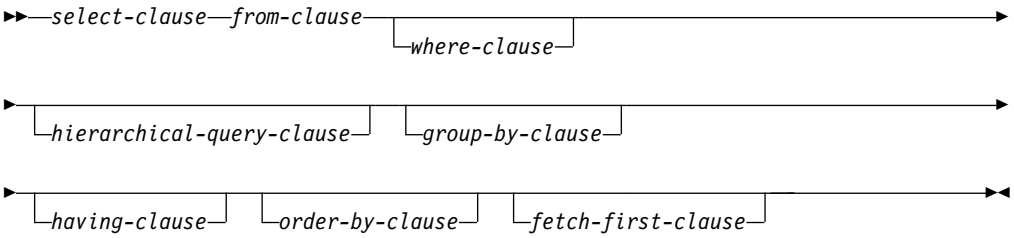
If a query includes a user-defined function, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each user-defined function identified in the statement:
  - The EXECUTE privilege on the function
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View or Corresponding System Authorities When Checking Privileges to a Function or Procedure.

subselect

The *subselect* is a component of the fullselect.



A subselect specifies a result table derived from the tables or views identified in the FROM clause. The derivation can be described as a sequence of operations in which the result of each operation is input for the next. (This is only a way of describing the subselect. The method used to perform the derivation may be quite different from this description. If portions of the subselect do not actually need to be executed for the correct result to be obtained, they may or may not be executed.)

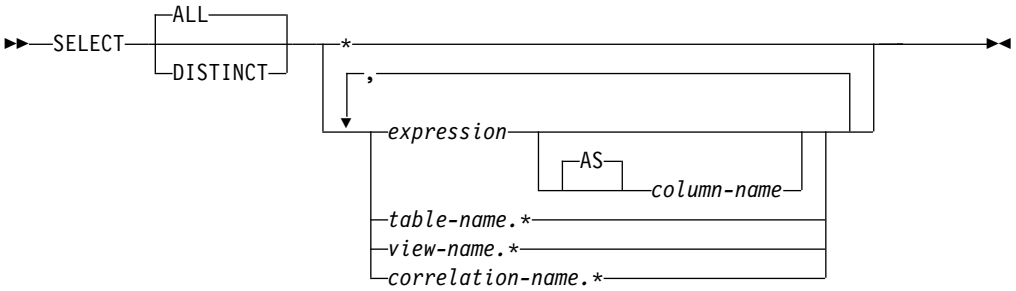
A *scalar-subselect* is a subselect, enclosed in parentheses, that returns a single result row and a single result column. If the result of the subselect is no rows, then the null value is returned. An error is returned if there is more than one row in the result.

The sequence of the (hypothetical) operations is:

1. FROM clause
2. hierarchical-query clause
3. WHERE clause
4. GROUP BY clause
5. HAVING clause
6. SELECT clause
7. ORDER BY clause
8. FETCH FIRST clause

select-clause

The SELECT clause specifies the columns of the final result table.



The column values are produced by the application of the *select list* to R. The select list is the names or expressions specified in the SELECT clause, and R is the result of the previous operation of the subselect. For example, if the only clauses specified are SELECT, FROM, and WHERE, R is the result of that WHERE clause.

ALL
Selects all rows of the final result table and does not eliminate duplicates. This is the default.

DISTINCT
Eliminates all but one of each set of duplicate rows of the final result table. Two rows are duplicates of one another only if each value in the first row is equal to the corresponding value in the second row. (For determining duplicate rows, two null values are considered equal.) The collating sequence is also used for determining distinct values.

DISTINCT is not allowed if the *select-list* contains a DATALINK or XML column, or an expression that returns a value that is the XML data type.

Select list notation

***** Represents a list of columns of table R in the order the columns are produced by the FROM clause. Any columns defined with the hidden attribute will not be included. The list of names is established when the statement containing the SELECT clause is prepared. Therefore, * does not identify any columns that have been added to a table after the statement has been prepared.

expression
Specifies the values of a result column. Each *column-name* in the *expression* must unambiguously identify a column of R.

column-name **or AS** *column-name*
Names or renames the result column. The name must not be qualified and does not have to be unique.

*name.**
Represents a list of columns of *name*. Any columns defined with the hidden attribute are not included. The *name* can be a table name, view name, or correlation name, and must designate an exposed table, view, or correlation name in the FROM clause immediately following the SELECT clause. The first name in the list identifies the first column of the table or view, the second name in the list identifies the second column of the table or view, and so on.

The list of names is established when the statement containing the SELECT clause is prepared. Therefore, * does not identify any columns that have been added to a table after the statement has been prepared.

Normally, when SQL statements are implicitly rebound, the list of names is not reestablished. Therefore, the number of columns returned by the statement does not change. However, there are four cases where the list of names is established again and the number of columns can change:

- When an SQL program or SQL package is saved and then restored on a IBM i product that is not the same release as the system from which it was saved.
- When SQL naming is specified for an SQL program or package and the owner of the program has changed since the SQL program or package was created.
- When an SQL statement is executed for the first time after the install of a more recent release of the IBM i operating system.
- When the SELECT * occurs in the fullselect of an INSERT statement or in a fullselect within a predicate, and a table or view referenced in the fullselect has been deleted and recreated with additional columns.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established at prepare time), and cannot exceed 8000. The result of a subquery must be a single expression, unless the subquery is used in the EXISTS predicate.

Applying the select list

The results of applying the select list to R depend on whether GROUP BY or HAVING is used:

If GROUP BY or HAVING is used

- Each *column-name* in the select list must identify a grouping expression, or be specified within an aggregate function, or be a correlated reference:
  - If the grouping expression is a column name, the select list may apply additional operators to the column name. For example, if the grouping expression is column C1, the select list may contain C1+1.
  - If the grouping expression is not a column name, the select list may not apply additional operators to the expression. For example, if the grouping expression is C1+1, the select list may contain C1+1, but not (C1+1)/8.
- The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the aggregate functions in the select list.
- The RRN, RID, DATAPARTITIONNAME, DATAPARTITIONNUM, DBPARTITIONNAME, DBPARTITIONNUM, and HASHED_VALUE functions cannot be specified in the select list.

If neither GROUP BY nor HAVING is used

- The select list must not include any aggregate functions, or each *column-name* must be specified in an aggregate function or be a correlated reference.
- If the select list does not include aggregate functions, it is applied to each row of R and the result contains as many rows as there are rows in R.
- If the select list is a list of aggregate functions, R is the source of the arguments of the functions and the result of applying the select list is one row.

In either case the *n*th column of the result contains the values specified by applying the *n*th expression in the operational form of the select list.

Null attributes of result columns

Result columns allow null values if they are derived from:

- Any aggregate function but COUNT and COUNT_BIG
- Any column that allows null values
- A scalar function or expression with an operand that allows null values
- A host variable that has an indicator variable, an SQL parameter or variable, or in the case of Java, a variable or expression whose type is able to represent a Java null value
- A result of a UNION or INTERSECT if at least one of the corresponding items in the select list is nullable
- An arithmetic expression in the outer select list
- A *scalar-fullselect*
- A user-defined scalar or table function
- A GROUPING SETS *grouping-expression*

Names of result columns

- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single column (without any functions or operators), then the result column name is the unqualified name of that column.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single variable (without any functions or operators), then the result column name is the unqualified name of that variable.
- All other result columns are unnamed.

Data types of result columns

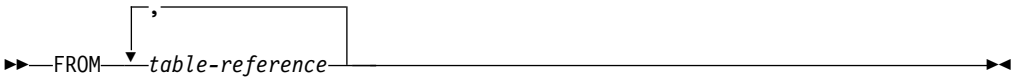
Each column of the result of SELECT acquires a data type from the expression from which it is derived.

When the expression is:	The data type of the result column is:
the name of any numeric column	the same as the data type of the column, with the same precision and scale for decimal columns.
an integer constant	INTEGER or BIGINT (if the value of the constant is outside the range of INTEGER, but within the range of BIGINT).
a decimal or floating-point constant	the same as the data type of the constant, with the same precision and scale for decimal constants.
the name of a DECFLOAT(7) variable	DECFLOAT(16)
the name of any numeric variable	the same as the data type of the variable, with the same precision and scale for decimal variables. If the data type of the variable is not identical to an SQL data type (for example, DISPLAY SIGN LEADING SEPARATE in COBOL), the result column is decimal.

When the expression is:	The data type of the result column is:
an expression	the same as the data type of the result, with the same precision and scale for decimal results as described under "Expressions" on page 165.
any function	the data type of the result of the function. For a built-in function, see Chapter 3, "Built-in functions," on page 227 to determine the data type of the result. For a user-defined function, the data type of the result is what was defined in the CREATE FUNCTION statement for the function.
the name of any string column	the same as the data type of the column, with the same length attribute.
the name of any string variable	the same as the data type of the variable, with a length attribute equal to the length of the variable. If the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string.
a character-string constant of length *n*	VARCHAR(*n*)
a graphic-string constant of length *n*	VARGRAPHIC(*n*)
the name of an XML column or variable	XML
the name of a datetime column, or an ILE RPG compiler or an ILE COBOL compiler datetime host variable	the same as the data type of the column or variable.
the name of a distinct type column	the same as the distinct type of the column, with the same length, precision, and scale attributes, if any.
the name of a datalink column	a datalink, with the same length attribute.
the name of a row ID column or a row ID variable	ROWID

from-clause

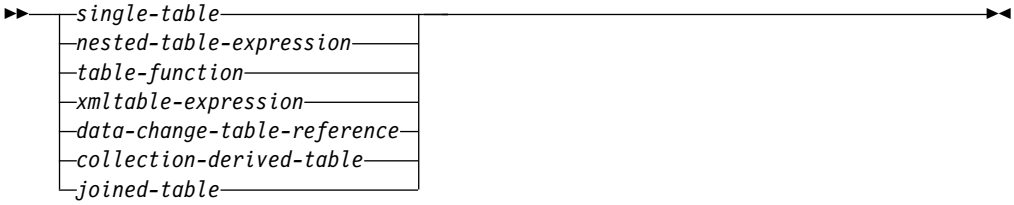
The FROM clause specifies an intermediate result table.



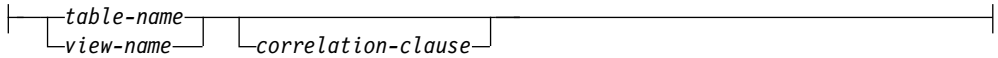
If only one *table-reference* is specified, the intermediate result table is simply the result of that *table-reference*. If more than one *table-reference* is specified in the FROM clause, the intermediate result table consists of all possible combinations of the rows of the specified *table-references* (the Cartesian product). Each row of the result is a row from the first *table-reference* concatenated with a row from the second *table-reference*, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the individual *table-references*.

table-reference

A *table-reference* specifies an intermediate result table.



single-table:



nested-table-expression:

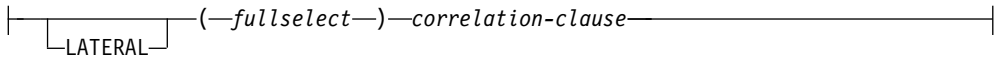
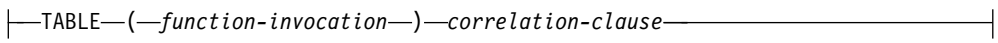
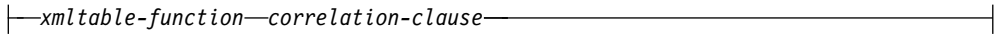


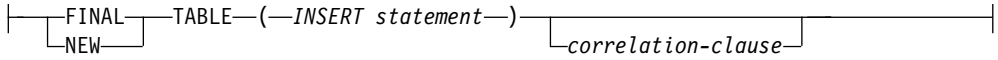
table-function:



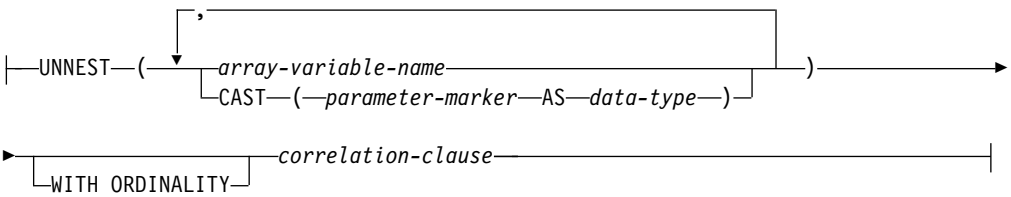
xmltable-expression:



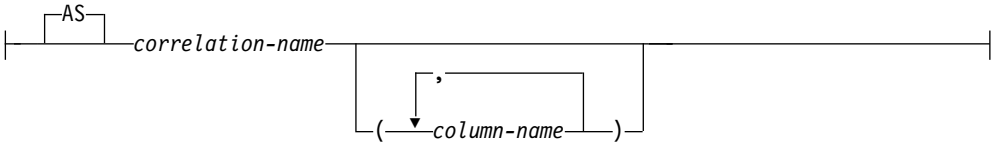
data-change-table-reference:



collection-derived-table:



correlation-clause:



- If a single table or view is identified, the intermediate result table is simply that table or view.
- A fullselect in parentheses called a *nested table expression*.[78] If a nested table expression is specified, the result table is the result of that nested table expression. The columns of the result do not need unique names, but a column with a non-unique name cannot be explicitly referenced.
- If a *function-name* or *xmltable-expression* is specified, the intermediate result table is the set of rows returned by the table function.
- If a *data-change-table-reference* is specified, the intermediate result table is the set of rows inserted by the INSERT statement.
- If a *collection-derived-table* is specified, the intermediate result table is a set of rows from one or more arrays.
- If a *joined-table* is specified, the intermediate result table is the result of one or more join operations. For more information, see "joined-table" on page 639.

If *table-reference* identifies a distributed table or a table that has a read trigger, the query cannot contain:

- EXCEPT or INTERSECT operations,
- VALUES in a fullselect,
- OLAP specifications,
- recursive common table expressions,
- ORDER OF,
- scalar fullselects (scalar subselects are supported),
- full outer join,
- LOBs in a GROUP BY,
- grouping sets or super groups,
- ORDER BY or FETCH FIRST n ROWS clause in a subselect,
- CONTAINS or SCORE functions,

78. A *nested table expression* is also called a *derived table*.

- XMLAGG, XMLATTRIBUTES, XMLCOMMENT, XMLCONCAT, XMLDOCUMENT, XMLELEMENT, XMLFOREST, XMLGROUP, XMLNAMESPACES, XMLPI, XMLROW, XMLTABLE, or XMLTEXT functions,
- global variables, or
- references to arrays.

The list of names in the FROM clause must conform to these rules:

- Each *table-name* and *view-name* must name an existing table or view at the current server or the *table-identifier* of a common table expression defined preceding the subselect containing the *table-reference*.
- The exposed names must be unique. An exposed name is a *correlation-name*, a *table-name* that is not followed by a *correlation-name*, or a *view-name* that is not followed by a *correlation-name*.
- Each *function-name*, together with the types of its arguments, must resolve to a table function that exists at the current server. An algorithm called function resolution, which is described on "Function resolution" on page 160, uses the function name and the arguments to determine the exact function to use. Unless given column names in the *correlation-clause*, the column names for a table function are those specified on the RETURNS clause of the CREATE FUNCTION statement. This is analogous to the column names of a table, which are defined in the CREATE TABLE statement.
- Each *array-variable-name* must identify an array variable in the SQL procedure.

Each *correlation-name* is defined as a designator of the intermediate result table specified by the immediately preceding *table-reference*. A *correlation-name* must be specified for nested table expressions and table functions.

The exposed names of all table references should be unique. An exposed name is:

- A *correlation-name*
- A *table-name* or *view-name* that is not followed by a *correlation-name*
- The *table-name* or *view-name* that is the target of the *data-change-table-reference* when the *data-change-table-reference* is not followed by a *correlation-name*

Any qualified reference to a column for a table, view, nested table expression, table function, *collection-derived-table*, or *data-change-table-reference* must use the exposed name. If the same table name or view name is specified twice, at least one specification should be followed by a *correlation-name*. The *correlation-name* is used to qualify references to the columns of the table or view. When a *correlation-name* is specified, column-names can also be specified to give names to the columns of the *table-name*, *view-name*, *nested-table-expression*, *table-function*, *data-change-table-reference*, or *collection-derived-table*. If a column list is specified, there must be a name in the column list for each column in the table or view and for each result column in the *nested-table-expression*, *table-function*, *data-change-table-reference*, or *collection-derived-table*. For more information, see "Correlation names" on page 140.

In general, *nested-table-expressions*, *table-functions*, and *collection-derived-tables* can be specified in any FROM clause. Columns from the nested table expressions, table functions, and collection derived tables can be referenced in the select list and in the rest of the subselect using the correlation name which must be specified. The scope of this correlation name is the same as correlation names for other table or view names in the FROM clause.

A nested table expression can be used:

- in place of a view to avoid creating the view (when general use of the view is not required)
- when the wanted result table is based on variables.

xmltable-function

Specifies an invocation of the built-in XMLTABLE table function. See "XMLTABLE" on page 604 for more information.

Data change table references

A *data-change-table-reference* specifies an intermediate result table that is based on the rows that are directly changed by the INSERT statement included in the clause. A *data-change-table-reference* must be the only *table-reference* in the FROM clause of the outer fullselect that is used in a select-statement, a SELECT INTO statement, a SET *variable* statement, or as the only fullselect in an assignment statement.

The intermediate result table for a *data-change-table-reference* includes all rows that were inserted. All columns of the inserted table may be referenced in the subselect, along with any INCLUDE columns defined on the INSERT statement. A *data-change-table-reference* has the following restrictions:

- It can appear only in the outer level fullselect.
- The target table or view of the INSERT statement is considered a table or view referenced in the query. Therefore, the authorization ID of the query must be authorized to the table or view as well as having the necessary privileges required by the INSERT.
- A fullselect in the INSERT statement cannot contain correlated references to columns outside the fullselect of the INSERT statement.
- A *data-change-table-reference* in a select-statement makes the cursor READ ONLY. This means that UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF cannot be used.
- If the INSERT references a view, the view must be defined using WITH CASCADED CHECK OPTION or could have been defined using WITH CHECK OPTION. In addition, the view cannot have a WHERE clause that contains:
  - a function that modifies SQL data
  - a function that is not deterministic or has external action
- A *data-change-table-reference* clause cannot be specified in a view definition or a materialized query table definition.
- If the target of the SQL data change statement is a view that is defined with an INSTEAD OF INSERT trigger, an error is returned.

FINAL TABLE
Specifies that the rows of the intermediate result table represent the set of rows that are inserted by the SQL data change statement as they appear at the completion of the data change statement. If there are AFTER INSERT triggers or referential constraints that result in further changes to the inserted rows of the table that is the target of the data change statement, an error is returned.

NEW TABLE
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement prior to the application of referential constraints and AFTER triggers. Data in the target table at the

completion of the statement might not match the data in the intermediate result table because of additional processing for referential constraints and AFTER triggers.

Collection derived table

A collection derived table can be used to unnest the elements of arrays into rows.

If more than one array is specified, the first array provides the first column in the result table, the second array provides the second column, and so on. If WITH ORDINALITY is specified, an extra column of type BIGINT, which contains the position of the elements in the arrays, is appended. If the cardinalities of the arrays are not identical, the cardinality of the result table is the same as the array with the largest cardinality. The column values in the table are set to the null value for all rows whose subindex value is greater than the cardinality of the corresponding array. In other words, if each array is viewed as a table with two columns (one for the subindices and one for the data), then UNNEST performs an OUTER JOIN among the arrays using equality on the subindices as the join predicate.

UNNEST can only be specified within an SQL procedure.

Correlated references in table-references

Correlated references can be used in *nested-table-expressions*. The basic rule that applies is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries. This hierarchy includes the *table-references* that have already been resolved in the left-to-right processing of the FROM clause. For nested table expressions, the TABLE or LATERAL keyword must appear before the fullselect. For more information see "References to SQL parameters and SQL variables" on page 1429

A table function can contain one or more correlated references to other tables in the same FROM clause if the referenced tables precede the reference in the left-to-right order of the tables in the FROM clause. The same capability exists for nested table expressions if the optional keyword TABLE or LATERAL is specified. Otherwise, only references to higher levels in the hierarchy of subqueries is allowed.

A nested table expression or table function that contains correlated references to other tables in the same FROM clause:

- Cannot participate in a RIGHT OUTER JOIN, FULL OUTER JOIN, or RIGHT EXCEPTION JOIN
- Can participate in LEFT OUTER JOIN or an INNER JOIN if the referenced tables precede the reference in the left-to-right order of the tables in the FROM clause

If *table-reference* identifies a distributed table or a table that has a read trigger; a nested table expression cannot contain a correlated reference to other tables in the same FROM clause when:

- The nested table expression contains a UNION, EXCEPT, or INTERSECT.
- The nested table expression uses the DISTINCT keyword in the select list.
- The nested table expression contains an ORDER BY and FETCH FIRST clause.
- The nested table expression is in the FROM clause of another nested table expression that contains one of these restrictions.

Syntax Alternatives: TABLE can be specified in place of LATERAL.

Example 1

The following example is valid:

```
SELECT D.DEPTNO, D.DEPTNAME, EMPINFO.AVGSAL, EMPINFO.EMPCOUNT
FROM DEPARTMENT D,
  (SELECT AVG(E.SALARY) AS AVGSAL,COUNT (*) AS EMPCOUNT
     FROM EMPLOYEE E
     WHERE E.WORKDEPT =
       (SELECT X.DEPTNO
         FROM DEPARTMENT X
         WHERE X.DEPTNO = E.WORKDEPT ) ) AS EMPINFO
```

The following example is not valid because the reference to D.DEPTNO in the WHERE clause of the *nested-table-expression* attempts to reference a table that is outside the hierarchy of subqueries:

```
SELECT D.DEPTNO, D.DEPTNAME,
        EMPINFO.AVGSAL, EMPINFO.EMPCOUNT                      ***INCORRECT***
  FROM DEPARTMENT D,
  (SELECT AVG(E.SALARY) AS AVGSAL,COUNT (*) AS EMPCOUNT
     FROM EMPLOYEE E
     WHERE E.WORKDEPT = D.DEPTNO ) AS EMPINFO
```

The following example is valid because the reference to D.DEPTNO in the WHERE clause of the *nested-table-expression* references DEPT, which precedes the *nested-table-expression* and the LATERAL keyword was specified:

```
SELECT D.DEPTNO, D.DEPTNAME,
        EMPINFO.AVGSAL, EMPINFO.EMPCOUNT
  FROM DEPARTMENT D,
  LATERAL (SELECT AVG(E.SALARY) AS AVGSAL,COUNT (*) AS EMPCOUNT
     FROM EMPLOYEE E
     WHERE E.WORKDEPT = D.DEPTNO ) AS EMPINFO
```

Example 2

The following example of a table function is valid:

```
SELECT t.c1, z.c5
  FROM t, TABLE(tf3 (t.c2 ) ) AS z
  WHERE t.c3 = z.c4
```

The following example is not valid because the reference to t.c2 is for a table that is to the right of the table function in the FROM clause:

```
SELECT t.c1, z.c5
  FROM TABLE(tf6 (t.c2 ) ) AS z, t                  ***INCORRECT***
  WHERE t.c3 = z.c4
```

Example 3

The following example of a table function is valid:

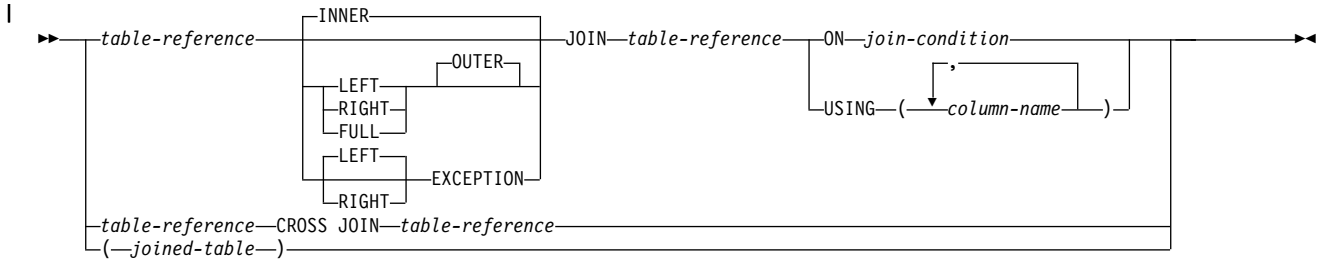
```
SELECT t.c1, z.c5
  FROM t, TABLE(tf4 (2 * t.c2 ) ) AS z
  WHERE t.c3 = z.c4
```

The following example is not valid because the reference to b.c2 is for the table function that is to the right of the table function containing the reference to b.c2 in the FROM clause:

```
SELECT a.c1, b.c5
  FROM TABLE(tf7a (b.c2 ) ) AS z,                   ***INCORRECT***
       TABLE(tf7b (a.c6 ) ) AS b
  WHERE a.c3 = b.c4
```

joined-table

A *joined-table* specifies an intermediate result table that is the result of either an inner, outer, cross, or exception join. The table is derived by applying one of the join operators: INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER, LEFT EXCEPTION, RIGHT EXCEPTION, or CROSS to its operands.



If a join operator is not specified, INNER is implicit. The order in which multiple joins are performed can affect the result. Joins can be nested within other joins. The order of processing for joins is generally from left to right, but based on the position of the required *join-condition* or USING clause. Parentheses are recommended to make the order of nested joins more readable. For example:

```
TB1 LEFT JOIN TB2 ON TB1.C1=TB2.C1
    LEFT JOIN TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1
           ON TB1.C1=TB3.C1
```

is the same as

```
(TB1 LEFT JOIN TB2 ON TB1.C1=TB2.C1)
     LEFT JOIN (TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1)
           ON TB1.C1=TB3.C1
```

An inner join combines each row of the left table with every row of the right table keeping only the rows where the *join-condition* (or USING clause) is true. Thus, the result table may be missing rows of from either or both of the joined tables. Outer joins include the rows produced by the inner join as well as the missing rows, depending on the type of outer join. Exception joins include only the missing rows, depending on the type of exception join.

- A left outer join includes the rows from the left table that were missing from the inner join.
- A right outer join includes the rows from the right table that were missing from the inner join.
- A full outer join includes the rows from both tables that were missing from the inner join.
- A left exception join includes only the rows from the left table that were missing from the inner join.
- A right exception join includes only the rows from the right table that were missing from the inner join.

A joined table can be used in any context in which any form of the SELECT statement is used. A view or a cursor is read-only if its SELECT statement includes a joined table.

Join condition

The *join-condition* is a *search-condition* that must conform to these rules:

- It cannot contain a quantified subquery, IN predicate with a subselect, or EXISTS subquery. It can contain basic predicate subqueries and scalar-fullselects.
- Any column referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join (in the scope of the same joined-table clause).
- Each column name must unambiguously identify a column in one of the tables in the *from-clause*.
- Aggregate functions cannot be used in the *expression*.

For any type of join, column references in an expression of the *join-condition* are resolved using the rules for resolution of column name qualifiers specified in "Column names" on page 140 before any rules about which tables the columns must belong to are applied.

Join USING

The USING clause specifies a shorthand way of defining the join condition. This form is known as a *named-columns-join*.

column-name
Must unambiguously identify a column that exists in both *table-references* of the joined table. The column must not be a DATALINK column.

The result table of the join contains the columns from the USING clause first, then the columns from the first table of the join that were not in the USING clause, followed by the remaining columns from the second table of the join that were not in the USING clause. Any column specified in the USING clause cannot be qualified in the query.

The USING clause is equivalent to a *join-condition* in which each column from the left *table-reference* is compared equal to a column of the same name in the right *table-reference*. For example, assume that TB1 and TB2 have columns C1, C2, ... Cn, D1, D2 *named-columns-join* of the form:

```
TB1 INNER JOIN TB2
    USING (C1, C2, ... Cn)
```

defines a result table that is equivalent to:

```
SELECT TB1.*, TB2.D1, TB2.D2
FROM TB1 INNER JOIN TB2
    ON TB1.C1 = TB2.C1 AND
       TB1.C2 = TB2.C2 AND
       ...
       TB1.Cn = TB2.Cn
```

Join operations

A *join-condition* (or USING clause) specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of the JOIN operator of the *join-condition* (or USING clause). For all possible combinations of rows of T1 and T2, a row of T1 is paired with a row of T2 if the *join-condition* (or USING clause) is true. When a row of T1 is joined with a row of T2, a row in the result consists of the values of that row of T1 concatenated with the values of that row of T2. In the case of OUTER joins, the execution might involve the generation of a null row. The null row of a table consists of a null value for each column of the table, regardless of whether the columns allow null values.

INNER JOIN or JOIN
The result of T1 INNER JOIN T2 consists of their paired rows.

Using the INNER JOIN syntax with a *join-condition* (or USING clause) will produce the same result as specifying the join by listing two tables in the FROM clause separated by commas and using the *where-clause* to provide the join condition.

LEFT JOIN or LEFT OUTER JOIN
The result of T1 LEFT OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T2 allow null values.

RIGHT JOIN or RIGHT OUTER JOIN
The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1. All columns derived from T1 allow null values.

FULL JOIN or FULL OUTER JOIN
The result of T1 FULL OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1 and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T1 and T2 allow null values.

FULL OUTER JOIN is not allowed if the query specifies:

- a distributed table,
- a table with a read trigger, or
- a logical file built over multiple physical file members.

LEFT EXCEPTION JOIN and EXCEPTION JOIN
The result of T1 LEFT EXCEPTION JOIN T2 consists only of each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T2 allow null values.

RIGHT EXCEPTION JOIN
The result of T1 RIGHT EXCEPTION JOIN T2 consists only of each unpaired row of T2, the concatenation of that row with the null row of T1. All columns derived from T1 allow null values.

CROSS JOIN
The result of T1 CROSS JOIN T2 consists of each row of T1 paired with each row of T2. CROSS JOIN is also known as Cartesian product.

Hierarchical queries

Hierarchical queries are a form of recursive query that provides support for retrieving a hierarchy, such as a bill of materials, from relational data using a CONNECT BY clause.

Connect-by recursion uses the same subquery for the seed (start) and the recursive step (connect). This combination provides a concise method of representing recursions such as bills-of-material, reports-to-chains, or email threads.

Connect-by recursion returns an error if a cycle occurs. A cycle occurs when a row produces itself, either directly or indirectly. Using the optional CONNECT BY NOCYCLE clause, the recursion can be directed to ignore the duplicated row, thus avoiding both the cycle and the error.

subselect

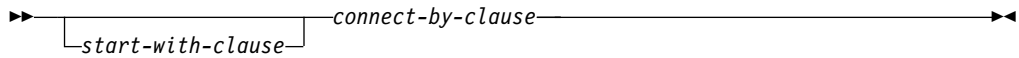
Hierarchical query support includes the following extensions to the subselect.

- The subselect includes a *hierarchical-query-clause*
- The clauses of the subselect are processed in the following sequence:
  1. FROM clause
  2. *hierarchical-query-clause*
  3. WHERE clause
  4. GROUP BY clause
  5. HAVING clause
  6. SELECT clause
  7. ORDER BY clause
  8. FETCH FIRST clause
- If the subselect includes a *hierarchical-query-clause*, special rules apply for the order of processing the predicates in the WHERE clause. The *search-condition* is factored into predicates along with its AND conditions (conjunction). If a predicate is an implicit join predicate (that is, it references more than one table in the FROM clause), the predicate is applied before the *hierarchical-query-clause*. Any predicate referencing at most one table in the FROM clause is applied to the intermediate result table of the *hierarchical-query-clause*.

  A hierarchical query involving joins should be written using explicit joined tables with an ON clause to avoid confusion about the application of WHERE clause predicates.
- The ORDER SIBLINGS BY clause can be specified if the subselect includes a *hierarchical-query-clause*. This clause specifies that the ordering applies only to siblings within the hierarchies.

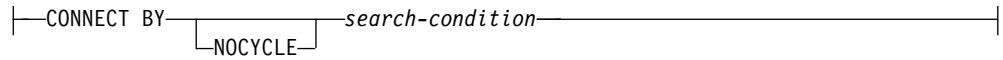
hierarchical-query-clause

A subselect that includes a *hierarchical-query-clause* is called a hierarchical query.



start-with-clause:

connect-by-clause:



After establishing a first intermediate result table H_1, subsequent intermediate result tables H_2, H_3, and so forth are generated by joining H_n with R using the *connect-by-clause* as a join condition to produce $H_{n+}1$. R is the result of the FROM clause of the subselect and any join predicates in the WHERE clause. The process stops when H_{n+1} has yielded an empty result table or when processing has reached the maximum supported depth level of 250. The result table H of the *hierarchical-query-clause* is the UNION ALL of every H_i.

start-with-clause
Specifies the intermediate result table H_1 for the hierarchical query that consists of those rows of R for which the *search-condition* is true. If the *start-with-clause* is not specified, H_1 is the entire intermediate result table R.

connect-by-clause
Produces the intermediate result table H_{n+1} from H_n by joining H_n with R, using the *search-condition*.

If the intermediate result table H_{n+1} would return a row from R for a hierarchical path that is the same as a row from R that is already in that hierarchical path, an error is returned.

NOCYCLE
Specifies that an error is not returned, but the repeated row is not included in the intermediate result table H_{n+1}.

The rules for the *search-condition* in the *start-with-clause* and the *connect-by-clause* are the same as for the *where-clause*.

The unary operator PRIOR is used to distinguish column references to H_n, the last prior recursive step, from column references to R. For example:

```
CONNECT BY MGRID = PRIOR EMPID
```

MGRID is resolved with R, and EMPID is resolved within the previous intermediate result table H_n.

A subselect that is a hierarchical query returns the intermediate result set in a partial order unless that order is destroyed through the use of an explicit ORDER BY clause, a GROUP BY or HAVING clause, or a DISTINCT keyword in the select list. The partial order returns rows, such that rows produced in H_{n+1} for a given

hierarchy immediately follow the row in H_n that produced them. The ORDER SIBLINGS BY clause can be used to enforce order within a set of rows produced by the same parent.

A NEXT VALUE expression cannot be specified in:

- The parameter list of a CONNECT_BY_ROOT operator or a SYS_CONNECT_BY_PATH function
- The START WITH or CONNECT BY clauses

Examples

- The following reports-to-chain example illustrates connect-by recursion. The example is based on a table named MY_EMP, which is created and populated with data as follows:

```
CREATE TABLE MY_EMP(
  EMPID  INTEGER NOT NULL PRIMARY KEY,
  NAME   VARCHAR(10),
  SALARY DECIMAL(9, 2),
  MGRID  INTEGER);

INSERT INTO MY_EMP VALUES ( 1, 'Jones',    30000, 10);
INSERT INTO MY_EMP VALUES ( 2, 'Hall',     35000, 10);
INSERT INTO MY_EMP VALUES ( 3, 'Kim',      40000, 10);
INSERT INTO MY_EMP VALUES ( 4, 'Lindsay',  38000, 10);
INSERT INTO MY_EMP VALUES ( 5, 'McKeough', 42000, 11);
INSERT INTO MY_EMP VALUES ( 6, 'Barnes',   41000, 11);
INSERT INTO MY_EMP VALUES ( 7, 'O''Neil',  36000, 12);
INSERT INTO MY_EMP VALUES ( 8, 'Smith',    34000, 12);
INSERT INTO MY_EMP VALUES ( 9, 'Shoeman',  33000, 12);
INSERT INTO MY_EMP VALUES (10, 'Monroe',   50000, 15);
INSERT INTO MY_EMP VALUES (11, 'Zander',   52000, 16);
INSERT INTO MY_EMP VALUES (12, 'Henry',    51000, 16);
INSERT INTO MY_EMP VALUES (13, 'Aaron',    54000, 15);
INSERT INTO MY_EMP VALUES (14, 'Scott',    53000, 16);
INSERT INTO MY_EMP VALUES (15, 'Mills',    70000, 17);
INSERT INTO MY_EMP VALUES (16, 'Goyal',    80000, 17);
INSERT INTO MY_EMP VALUES (17, 'Urbassek', 95000, NULL);
```

The following query returns all employees working for Goyal, as well as some additional information, such as the reports-to-chain:

```
1 SELECT NAME,
2        LEVEL,
3        SALARY,
4        CONNECT_BY_ROOT NAME AS ROOT,
5        SUBSTR(SYS_CONNECT_BY_PATH(NAME, ':'), 1, 25) AS CHAIN
6   FROM MY_EMP
7   START WITH NAME = 'Goyal'
8   CONNECT BY PRIOR EMPID = MGRID
9   ORDER SIBLINGS BY SALARY;
NAME       LEVEL       SALARY      ROOT  CHAIN
---------- ----------- ----------- ----- ---------------
Goyal                1    80000.00 Goyal :Goyal
Henry                2    51000.00 Goyal :Goyal:Henry
Shoeman              3    33000.00 Goyal :Goyal:Henry:Shoeman
Smith                3    34000.00 Goyal :Goyal:Henry:Smith
O'Neil               3    36000.00 Goyal :Goyal:Henry:O'Neil
Zander               2    52000.00 Goyal :Goyal:Zander
Barnes               3    41000.00 Goyal :Goyal:Zander:Barnes
McKeough             3    42000.00 Goyal :Goyal:Zander:McKeough
Scott                2    53000.00 Goyal :Goyal:Scott
```

Lines 7 and 8 comprise the core of the recursion. The optional START WITH clause describes the WHERE clause that is to be used on the source table to seed the recursion. In this case, only the row for employee Goyal is selected. If the

START WITH clause is omitted, the entire source table is used to seed the recursion. The CONNECT BY clause describes how, given the existing rows, the next set of rows is to be found. The unary operator PRIOR is used to distinguish values in the previous step from those in the current step. PRIOR identifies EMPID as the employee ID of the previous recursive step, and MGRID as originating from the current recursive step.

LEVEL in line 2 is a pseudo column that describes the current level of recursion.

CONNECT_BY_ROOT is a unary operator that always returns the value of its argument as it was during the first recursive step; that is, the values that are returned by an explicit or implicit START WITH clause.

SYS_CONNECT_BY_PATH() is a binary function that prepends the second argument to the first and then appends the result to the value that it produced in the previous recursive step.

Unless explicitly overridden, connect-by recursion returns a result set in a partial order; that is, the rows that are produced by a recursive step always follow the row that produced them. Siblings at the same level of recursion have no specific order. The ORDER SIBLINGS BY clause in line 9 defines an order for these siblings, which further refines the partial order, potentially into a total order.

- Return the organizational structure of the DEPARTMENT table. Use the level of the department to visualize the hierarchy.

```
SELECT LEVEL, CAST(SPACE((LEVEL - 1) * 4) || '/' || DEPTNAME
    AS VARCHAR(40)) AS DEPTNAME
  FROM DEPARTMENT
  START WITH DEPTNO = 'A00'
  CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

The query returns:

```
LEVEL       DEPTNAME
----------- ----------------------------------------
          1 /SPIFFY COMPUTER SERVICE DIV.
          2     /PLANNING
          2     /INFORMATION CENTER
          2     /DEVELOPMENT CENTER
          3         /MANUFACTURING SYSTEMS
          3         /ADMINISTRATION SYSTEMS
          2     /SUPPORT SERVICES
          3         /OPERATIONS
          3         /SOFTWARE SUPPORT
          3         /BRANCH OFFICE F2
          3         /BRANCH OFFICE G2
          3         /BRANCH OFFICE H2
          3         /BRANCH OFFICE I2
          3         /BRANCH OFFICE J2
```

pseudo columns

A pseudo column is an identifier that shares the same namespace as columns and variables. After regular name resolution for a column has failed, DB2 attempts to match an unqualified identifier to a pseudo column name.

CONNECT_BY_ISCYCLE

CONNECT_BY_ISCYCLE is a pseudo column for use in hierarchical queries. The column returns the value 1 if the row is part of a cycle in the hierarchy; that is, the row has itself as a direct or indirect ancestor given the CONNECT BY clause's *search-condition*. If the row is not part of a cycle, the column returns the value 0.

The data type of the column is SMALLINT and it cannot be null.

CONNECT_BY_ISCYCLE must be specified in the context of a hierarchical query but cannot be specified in the START WITH clause or the CONNECT BY clause, as an argument of the CONNECT_BY_ROOT operator, or as an argument of the SYS_CONNECT_BY_PATH function.

CONNECT_BY_ISLEAF

CONNECT_BY_ISLEAF is a pseudo column for use in hierarchical queries. The column returns the value 1 if the row is a leaf in the hierarchy as defined by the CONNECT BY clause. If the row is not a leaf, the column returns the value 0.

The data type of the column is SMALLINT and it cannot be null.

CONNECT_BY_ISLEAF must be specified in the context of a hierarchical query but cannot be specified in the START WITH clause or the CONNECT BY clause, as an argument of the CONNECT_BY_ROOT operator, or as an argument of the SYS_CONNECT_BY_PATH function.

LEVEL

LEVEL is a pseudo column for use in hierarchical queries. The column returns the recursive step in the hierarchy at which the row was produced. All rows produced by the START WITH clause return the value 1. Rows produced by applying the first iteration of the CONNECT BY clause return 2, and so on.

The data type of the column is INTEGER and it cannot be null.

LEVEL must be specified in the context of a hierarchical query but cannot be specified in the START WITH clause, as an argument of the CONNECT_BY_ROOT operator, or as an argument of the SYS_CONNECT_BY_PATH function.

CONNECT_BY_ROOT

The CONNECT_BY_ROOT unary operator is for use only in hierarchical queries. For every row in the hierarchy, this operator returns the expression for the row's root ancestor.

expression
An expression that does not contain a NEXT VALUE expression, a hierarchical query construct (such as the LEVEL pseudocolumn), the SYS_CONNECT_BY_PATH function, or an OLAP specification.

The result data type and length of the operator is the same as the result data type and length of the expression.

A CONNECT_BY_ROOT operator cannot be specified in the START WITH clause or the CONNECT BY clause of a hierarchical query. It cannot be specified as an argument to the SYS_CONNECT_BY_PATH function.

A CONNECT_BY_ROOT operator has a higher precedence than any infix operator. Therefore, to pass an expression with infix operators (such as + or ||) as an argument, parentheses must be used. For example:

```
CONNECT_BY_ROOT FIRSTNME || LASTNAME
```

returns the FIRSTNME value of the root ancestor row concatenated with the LASTNAME value of the actual row in the hierarchy, because this expression is equivalent to:

```
(CONNECT_BY_ROOT FIRSTNME) || LASTNAME
```

rather than:

```
CONNECT_BY_ROOT (FIRSTNME || LASTNAME)
```

Example

- Return the hierarchy of departments and their root departments in the DEPARTMENT table.

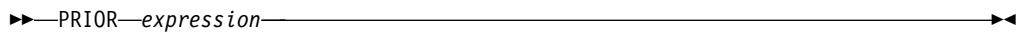
```
SELECT CONNECT_BY_ROOT DEPTNAME AS ROOT, DEPTNAME
     FROM DEPARTMENT
     START WITH DEPTNO IN ('B01','C01','D01','E01')
     CONNECT BY PRIOR DEPTNO = ADMRDEPT
```

This query returns:

```
ROOT                DEPTNAME
------------------- ----------------------
PLANNING            PLANNING
INFORMATION CENTER  INFORMATION CENTER
DEVELOPMENT CENTER  DEVELOPMENT CENTER
DEVELOPMENT CENTER  MANUFACTURING SYSTEMS
DEVELOPMENT CENTER  ADMINISTRATION SYSTEMS
SUPPORT SERVICES    SUPPORT SERVICES
SUPPORT SERVICES    OPERATIONS
SUPPORT SERVICES    SOFTWARE SUPPORT
SUPPORT SERVICES    BRANCH OFFICE F2
SUPPORT SERVICES    BRANCH OFFICE G2
SUPPORT SERVICES    BRANCH OFFICE H2
SUPPORT SERVICES    BRANCH OFFICE I2
SUPPORT SERVICES    BRANCH OFFICE J2
```

PRIOR

The PRIOR unary operator is for use only in the CONNECT BY clause of hierarchical queries.



The CONNECT BY clause performs an inner join between the intermediate result table H_n of the hierarchical query and the source result table specified in the FROM clause. All column references to tables that are referenced in the FROM clause, and which are arguments to the PRIOR operator, are considered to be ranging over H_n.

The primary key of the intermediate result table H_n is typically joined to the foreign keys of the source result table to recursively traverse the hierarchy.

```
CONNECT BY PRIOR T.PK = T.FK
```

If the primary key is a composite key, care must be taken to prefix each column with PRIOR:

```
CONNECT BY PRIOR T.PK1 = T.FK1 AND PRIOR T.PK2 = T.FK2
```

expression
An expression that does not contain a NEXT VALUE expression, a hierarchical query construct (such as the LEVEL pseudocolumn), the SYS_CONNECT_BY_PATH function, or an OLAP specification.

The result data type and length of the operator is the same as the result data type and length of the expression.

A PRIOR operator has a higher precedence than any infix operator. Therefore, to pass an expression with infix operators (such as + or ||) as an argument, parentheses must be used. For example:

```
PRIOR FIRSTNME || LASTNAME
```

returns the FIRSTNME value of the prior row concatenated with the LASTNAME value of the actual row in the hierarchy, because this expression is equivalent to:

```
(PRIOR FIRSTNME) || LASTNAME
```

rather than:

```
PRIOR (FIRSTNME || LASTNAME)
```

Example

- Return the hierarchy of departments in the DEPARTMENT table.

```
SELECT LEVEL, DEPTNAME
     FROM DEPARTMENT
     START WITH DEPTNO = 'A00'
     CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

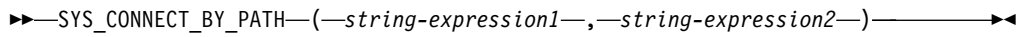
This query returns:

```
LEVEL       DEPTNAME
----------- ----------------------------------------
          1 SPIFFY COMPUTER SERVICE DIV.
          2 PLANNING
          2 INFORMATION CENTER
          2 DEVELOPMENT CENTER
          3 MANUFACTURING SYSTEMS
```

```
|                                3 ADMINISTRATION SYSTEMS
|                                2 SUPPORT SERVICES
|                                3 OPERATIONS
|                                3 SOFTWARE SUPPORT
|                                3 BRANCH OFFICE F2
|                                3 BRANCH OFFICE G2
|                                3 BRANCH OFFICE H2
|                                3 BRANCH OFFICE I2
|                                3 BRANCH OFFICE J2

|
```

SYS_CONNECT_BY_PATH

The SYS_CONNECT_BY_PATH function is used in hierarchical queries to build a string representing a path from the root row to this row.



The string for a given row at LEVEL *n* is built as follows:

- Step 1 (using the values of the root row from the first intermediate result table H_1):

  ```
  path1 := string-expression2 || string-expression1
  ```

- Step n (based on the row from the intermediate result table H_n):

  ```
  path_n := path_n-1 || string-expression2 || string-expression1
  ```

string-expression1
A character string expression that identifies the row. It must be a built-in character string data type. The expression must not include a NEXT VALUE expression for a sequence, any hierarchical query construct such as the LEVEL pseudocolumn or the CONNECT_BY_ROOT operator, an OLAP specification, or an aggregate function.

string-expression2
A character string expression that serves as a separator. It must be a built-in character string data type. The expression must not include a NEXT VALUE expression for a sequence, any hierarchical query construct such as the LEVEL pseudocolumn or the CONNECT_BY_ROOT operator, an OLAP specification, or an aggregate function.

The result is a CLOB(1M).

The SYS_CONNECT_BY_PATH function must not be used outside of the context of a hierarchical query. It cannot be used in a START WITH clause or a CONNECT BY clause.

Example

- Return the hierarchy of departments in the DEPARTMENT table.

```
SELECT CAST(SYS_CONNECT_BY_PATH(DEPTNAME, '/')
                  AS VARCHAR(76)) AS ORG
       FROM DEPARTMENT
       START WITH DEPTNO = 'A00'
       CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

This query returns:

```
ORG
------------------------------------------------------------------------
/SPIFFY COMPUTER SERVICE DIV.
/SPIFFY COMPUTER SERVICE DIV./PLANNING
/SPIFFY COMPUTER SERVICE DIV./INFORMATION CENTER
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER/MANUFACTURING SYSTEMS
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER/ADMINISTRATION SYSTEMS
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/OPERATIONS
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/SOFTWARE SUPPORT
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE F2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE G2
```

```
|                          /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE H2
|                          /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE I2
|                          /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE J2

|
```

where-clause

The WHERE clause specifies an intermediate result table that consists of those rows of R for which the *search-condition* is true. R is the result of the FROM clause of the statement.

The *search-condition* must conform to the following rules:

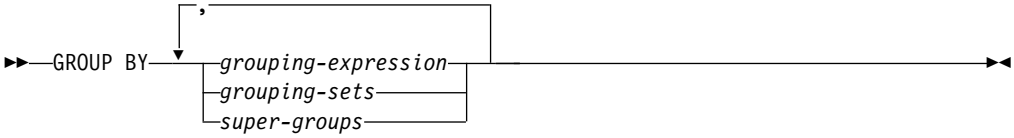
- Each *column-name* must unambiguously identify a column of R or be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a table, view, *common-table-expression*, or *nested-table-expression* identified in an outer fullselect.
- An aggregate function must not be specified unless the WHERE clause is specified in a subquery of a HAVING clause and the argument of the function is a correlated reference to a group.

Any subquery in the *search-condition* is effectively executed for each row of R and the results are used in the application of the *search-condition* to the given row of R. A subquery is executed for each row of R if it includes a correlated reference to a column of R. A subquery with no correlated reference is typically executed just once.

If a collating sequence other than *HEX is in effect when the statement that contains the WHERE clause is executed and if the *search-condition* contains operands that are SBCS data, mixed data, or Unicode data, then the comparison for those predicates is done using weighted values. The weighted values are derived by applying the collating sequence to the operands of the predicate.

group-by-clause

The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause of the subselect.



In its simplest form, a GROUP BY clause contains a *grouping-expression*. A *grouping-expression* is an expression that defines the grouping of R. The following restrictions apply to *grouping-expression*.

- Each column name included in *grouping-expression* must unambiguously identify a column of R.
- The result of *grouping-expression* cannot be a DataLink or XML data type or a distinct type that is based on a DataLink or XML.
- *grouping-expression* cannot include any of the following items:
  - A correlated column
  - A variable
  - An aggregate function
  - Any function that is non-deterministic

More complex forms of the GROUP BY clause include *grouping-sets* and *super-groups*. For a description of these forms, see "grouping-sets" on page 654 and "super-groups" on page 655, respectively.

The result of the GROUP BY clause is a set of groups of rows. In each group of more than one row, all values of each *grouping-expression* are equal, and all rows with the same set of values of the *grouping-expressions* are in the same group. For grouping, all null values for a *grouping-expression* are considered equal.

Because every row of a group contains the same value of any *grouping-expression*, *grouping-expressions* can be used in a search condition in a HAVING clause, in the SELECT clause, or in a *sort-key-expression* of an ORDER BY clause (see "order-by-clause" on page 668 for details). In each case, the reference specifies only one value for each group. The *grouping-expression* specified in these clauses must exactly match the *grouping-expression* in the GROUP BY clause, except that blanks are not significant. For example, a *grouping-expression* of

```
SALARY*.10
```

will match the expression in a *having-clause* of

```
HAVING SALARY*.10
```

but will not match

```
HAVING .10 *SALARY
 or
HAVING (SALARY*.10)+100
```

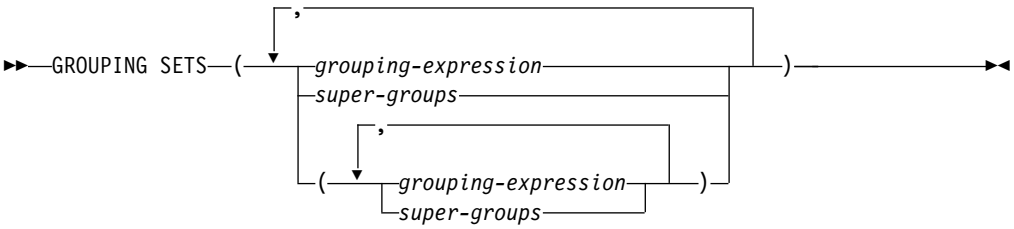
If the *grouping-expression* contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and may not all have the same length. In that case, a reference to the *grouping-expression* still

specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual length of the result value is unpredictable.

The sum of the length attributes of *grouping-expressions* must not exceed 32766-*n* bytes, where *n* is the number of *grouping-expressions* specified that allow nulls.

If a collating sequence other than *HEX is in effect when the statement that contains the GROUP BY clause is executed, and the *grouping-expressions* are SBCS data, mixed data, or Unicode data, then the rows are placed into groups using the weighted values. The weighted values are derived by applying the collating sequence to the *grouping-expressions*. In that case, a reference to the *grouping-expression* still specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual value of the result is unpredictable.

grouping-sets



A *grouping-sets* specification allows multiple grouping clauses to be specified in a single statement. This can be thought of as the union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple subselects with the group by clause in each subselect corresponding to one grouping set. A grouping set can be a single element or can be a list of elements delimited by parentheses, where an element is either a *grouping-expression* or a *super-group*. Using grouping sets allows the groups to be computed with a single pass over the base table.

The *grouping-sets* specification allows either a simple *grouping-expression* to be used, or the more complex forms of *super-groups*. For a description of *super-groups*, see "super-groups" on page 655.

Note that grouping sets are the fundamental building blocks for GROUP BY operations. A simple GROUP BY with a single column can be considered a grouping set with one element. For example:

```
GROUP BY a
```

is the same as

```
GROUP BY GROUPING SETS( (a) )
```

and

```
GROUP BY a, b, c
```

is the same as

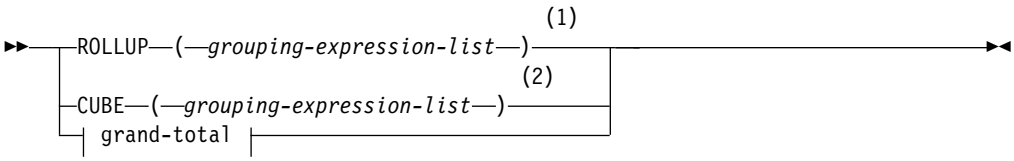
```
GROUP BY GROUPING SETS( (a,b,c) )
```

Non-aggregation columns from the select list of the subselect that are excluded from a grouping set will return a null for such columns for each row generated for that grouping set. This reflects the fact that aggregation was done without considering the values for those columns.

If a *table-reference* in the previous clauses of the query identifies a distributed table or a table that has a read trigger; a *grouping-sets* specification is not allowed.

Example C2 through Example C7 illustrate the use of grouping sets.

super-groups

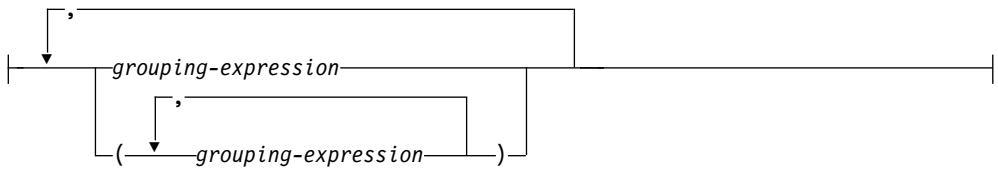


Notes:

1 Alternate specification when used alone in *group-by-clause* is: *grouping-expression-list* WITH ROLLUP.

2 Alternate specification when used alone in *group-by-clause* is: *grouping-expression-list* WITH CUBE.

grouping-expression-list:



grand-total:

ROLLUP (grouping-expression-list)

A *ROLLUP grouping* is an extension to the GROUP BY clause that produces a result set containing *sub-total* rows in addition to the "regular" grouped rows. *Sub-total* rows are "super-aggregate" rows that contain further aggregates whose values are derived by applying the same column functions that were used to obtain the grouped rows. These rows are called sub-total rows because that is their most common use; however, any aggregate function can be used for the aggregation. For instance, MAX and AVG are used in example C8.

A ROLLUP grouping is a series of *grouping-sets*. The general specification of a ROLLUP with *n* elements

GROUP BY ROLLUP(C_1,C_2,...,C_{n-1},C_n**)**

is equivalent to

```
GROUP BY GROUPING SETS( (C1,C2,...,Cn-1,Cn),
                        (C1,C2,...,Cn-1),
                        ...
                        (C1,C2),
                        (C1),
                        () )
```

Note that the *n* elements of the ROLLUP translate to *n*+1 grouping sets. Note also that the order in which the *grouping-expressions* are specified is significant for ROLLUP. For example:

```
GROUP BY ROLLUP (a,b)
```

is equivalent to

```
GROUP BY GROUPING SETS ( (a,b),
                         (a),
                         () )
```

while

```
GROUP BY ROLLUP (b,a)
```

is equivalent to

```
GROUP BY GROUPING SETS ( (b,a),
                         (b),
                         () )
```

The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example C3 illustrates the use of ROLLUP.

CUBE (*grouping-expression-list* **)**

A *CUBE grouping* is an extension to the GROUP BY clause that produces a result set that contains all the rows of a ROLLUP aggregation and, in addition, contains "cross-tabulation" rows. *Cross-tabulation* rows are additional "super-aggregate" rows that are not part of an aggregation with sub-totals. Only 10 expressions are allowed in the *grouping-expression-list*.

Like a ROLLUP, a CUBE grouping can also be thought of as a series of *grouping-sets*. In the case of a CUBE, all permutations of the cubed *grouping-expression-list* are computed along with the grand total. Therefore, the *n* elements of a CUBE translate to 2 ** *n* (2 to the power *n*) grouping-sets. For instance, a specification of

```
GROUP BY CUBE (a,b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS ( (a,b,c),
                         (a,b),
                         (a,c),
                         (b,c),
                         (a),
                         (b),
                         (c),
                         () )
```

Notice that the 3 elements of the CUBE translate to 8 grouping sets.

The order of specification of elements does not matter for CUBE. 'CUBE(DayOfYear, Sales_Person)' and 'CUBE(Sales_Person, DayOfYear) yield the same result sets. The use of the word 'same' applies to content of the result set, not to its order. The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example C4 illustrates the use of CUBE.

grouping-expression-list

A *grouping-expression-list* is used within a CUBE or ROLLUP grouping to define the number of elements in the CUBE or ROLLUP operation. This is controlled by using parentheses to delimit elements with multiple *grouping-expressions*.

The rules for a *grouping-expression* are described in "group-by-clause" on page 653. For example, suppose that a query is to return the total expenses for the ROLLUP of City within a Province but not within a County. However, the clause

```
GROUP BY ROLLUP (Province, County, City)
```

results in unwanted sub-total rows for the County. In the clause

```
GROUP BY ROLLUP (Province, (County, City))
```

the composite (County, City) forms one element in the ROLLUP and, therefore, a query that uses this clause will yield the wanted result. In other words, the two element ROLLUP

```
GROUP BY ROLLUP (Province, (County, City))
```

generates

```
GROUP BY GROUPING SETS ( (Province, County, City)
                         (Province)
                         () )
```

while the three element ROLLUP generates

```
GROUP BY GROUPING SETS ( (Province, County, City),
                         (Province, County),
                         (Province),
                         () )
```

Example C2 also uses composite column values.

grand-total

Both CUBE and ROLLUP return a row which is the overall (grand total) aggregation. This may be separately specified with empty parentheses within the GROUPING SETS clause. It may also be specified directly in the GROUP BY clause, although there is no effect on the result of the query. Example C4 uses the grand-total syntax.

Combining grouping sets

This can be used to combine any of the types of GROUP BY clauses. When simple *grouping-expressions* are combined with other groups, they are "appended" to the beginning of the resulting *grouping sets*. When ROLLUP and CUBE expressions are combined, they operate like "multipliers" on the remaining expression, forming additional grouping set entries according to the definition of either ROLLUP or CUBE.

For instance, combining *grouping-expression* elements acts as follows

```
GROUP BY a, ROLLUP (b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS( (a,b,c),
                        (a,b),
                        (a) )
```

Or similarly

```
GROUP BY a,b, ROLLUP (c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS( (a,b,c,d),
                        (a,b,c),
                        (a,b) )
```

Combining of ROLLUP elements acts as follows:

```
GROUP BY ROLLUP(a), ROLLUP (b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS( (a,b,c),
                        (a,b),
                        (a),
                        (b,c),
                        (b),
                        () )
```

Similarly,

```
GROUP BY ROLLUP(a), CUBE (b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS( (a,b,c),
                        (a,b),
                        (a,c),
                        (a),
                        (b,c),
                        (b),
                        (c),
                        () )
```

Combining of CUBE and ROLLUP elements acts as follows:

```
GROUP BY CUBE(a,b), ROLLUP (c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS( (a,b,c,d),
                        (a,b,c),
                        (a,b),
                        (a,c,d),
                        (a,c),
                        (a),
                        (b,c,d),
                        (b,c),
                        (b),
                        (c,d),
                        (c),
                        () )
```

Like a simple *grouping-expression*, combining grouping sets also eliminates duplicates within each grouping set. For instance,

```
GROUP BY a, ROLLUP (a,b)
```

is equivalent to

```
GROUP BY GROUPING SETS( (a,b),
                        (a) )
```

A more complete example of combining grouping sets is to construct a result set that eliminates certain rows that would be returned for a full CUBE aggregation.

For example, consider the following GROUP BY clause:

```
GROUP BY Region,
         ROLLUP (Sales_Person, WEEK(Sales_Date)),
         CUBE (YEAR(Sales_Date), MONTH(Sales_Date))
```

The column listed immediately to the right of GROUP BY is grouped, those within the parentheses following ROLLUP are rolled up, and those within the parentheses following CUBE are cubed. Thus, the above clause results in a cube of MONTH within YEAR which is then rolled up within WEEK within Sales_Person within the Region aggregation. It does not result in any grand total row or any cross-tabulation rows on Region, Sales_Person, or WEEK(Sales_Date) so produces fewer rows than the clause:

```
GROUP BY  ROLLUP (Region, Sales_Person, WEEK(Sales_Date),
                  YEAR(Sales_Date), MONTH(Sales_Date))
```

Examples of grouping sets, cube, and rollup

The queries in Example C1 through C4 use a subset of the rows in the SALES table based on the predicate 'WEEK(SALES_DATE) = 13'.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON,
       SALES AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
```

which results in:

```
WEEK    DAY_WEEK   SALES_PERSON  UNITS_SOLD
------- ---------- ------------- -----------
     13          6 LUCCHESSI               3
     13          6 LUCCHESSI               1
     13          6 LEE                     2
     13          6 LEE                     2
     13          6 LEE                     3
     13          6 LEE                     5
     13          6 GOUNOT                  3
     13          6 GOUNOT                  1
     13          6 GOUNOT                  7
     13          7 LUCCHESSI               1
     13          7 LUCCHESSI               2
     13          7 LUCCHESSI               1
     13          7 LEE                     7
     13          7 LEE                     3
     13          7 LEE                     7
     13          7 LEE                     4
     13          7 GOUNOT                  2
     13          7 GOUNOT                 18
     13          7 GOUNOT                  1
```

Example C1:

Here is a query with a basic GROUP BY over 3 columns:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK    DAY_WEEK   SALES_PERSON  UNITS_SOLD
------- ---------- ------------- ------------
     13          6 GOUNOT                 11
     13          6 LEE                    12
     13          6 LUCCHESSI               4
     13          7 GOUNOT                 21
     13          7 LEE                    21
     13          7 LUCCHESSI               4
```

Example C2:

Produce the result based on two different grouping sets of rows from the SALES table.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY GROUPING SETS( (WEEK(SALES_DATE), SALES_PERSON),
                        (DAYOFWEEK(SALES_DATE), SALES_PERSON) )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK    DAY_WEEK   SALES_PERSON  UNITS_SOLD
------- ---------- ------------- ------------
     13          - GOUNOT                 32
     13          - LEE                    33
     13          - LUCCHESSI               8
      -          6 GOUNOT                 11
      -          6 LEE                    12
      -          6 LUCCHESSI               4
      -          7 GOUNOT                 21
      -          7 LEE                    21
      -          7 LUCCHESSI               4
```

The rows with WEEK 13 are from the first grouping set and the other rows are from the second grouping set.

Example C3:

If you use 3 distinct columns involved in the grouping sets of Example C2 and perform a ROLLUP, you can see grouping sets for (WEEK, DAY_WEEK, SALES_PERSON), (WEEK, DAY_WEEK), (WEEK), and grand total.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK    DAY_WEEK   SALES_PERSON  UNITS_SOLD
------- ---------- ------------- ------------
     13          6 GOUNOT                 11
     13          6 LEE                    12
     13          6 LUCCHESSI               4
     13          6 -                      27
```

```
13          7 GOUNOT                 21
13          7 LEE                    21
13          7 LUCCHESSI               4
13          7 -                      46
13          - -                      73
 -          - -                      73
```

Example C4:

If you run the same query as Example C3 only replace ROLLUP with CUBE, you can see additional grouping sets for (WEEK, SALES_PERSON), (DAY_WEEK, SALES_PERSON), (DAY_WEEK), and (SALES_PERSON) in the result.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY CUBE( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK    DAY_WEEK   SALES_PERSON  UNITS_SOLD
------- ---------- ------------- -----------
     13          6 GOUNOT                11
     13          6 LEE                   12
     13          6 LUCCHESSI              4
     13          6 -                     27
     13          7 GOUNOT                21
     13          7 LEE                   21
     13          7 LUCCHESSI              4
     13          7 -                     46
     13          - GOUNOT                32
     13          - LEE                   33
     13          - LUCCHESSI              8
     13          - -                     73
      -          6 GOUNOT                11
      -          6 LEE                   12
      -          6 LUCCHESSI              4
      -          6 -                     27
      -          7 GOUNOT                21
      -          7 LEE                   21
      -          7 LUCCHESSI              4
      -          7 -                     46
      -          - GOUNOT                32
      -          - LEE                   33
      -          - LUCCHESSI              8
      -          - -                     73
```

Example C5:

Obtain a result set which includes a grand total of selected rows from the SALES table together with a group of rows aggregated by SALES_PERSON and MONTH.

```
SELECT SALES_PERSON,
       MONTH(SALES_DATE) AS MONTH,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS( (SALES_PERSON, MONTH(SALES_DATE)),
                                    () )
ORDER BY SALES_PERSON, MONTH
```

This results in:

```
SALES_PERSON  MONTH   UNITS_SOLD
------------- ------- ------------
GOUNOT              3           35
GOUNOT              4           14
GOUNOT             12            1
LEE                 3           60
LEE                 4           25
LEE                12            6
LUCCHESSI           3            9
LUCCHESSI           4            4
LUCCHESSI          12            1
-                   -          155
```

Example C6:

This example shows two simple ROLLUP queries followed by a query which treats the two ROLLUPs as grouping sets in a single result set and specifies row ordering for each column involved in the grouping sets.

Example C6-1:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) )
ORDER BY WEEK, DAY_WEEK
```

This results in:

```
WEEK     DAY_WEEK   UNITS_SOLD
-------- ---------- ------------
      13          6           27
      13          7           46
      13          -           73
      14          1           31
      14          2           43
      14          -           74
      53          1            8
      53          -            8
       -          -          155
```

Example C6-2:

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP( MONTH(SALES_DATE), REGION )
ORDER BY MONTH, REGION
```

This results in:

```
MONTH    REGION          UNITS_SOLD
-------- --------------  ------------
       3 Manitoba                 22
       3 Ontario-North             8
       3 Ontario-South            34
       3 Quebec                   40
       3 -                       104
       4 Manitoba                 17
       4 Ontario-North             1
       4 Ontario-South            14
       4 Quebec                   11
       4 -                        43
      12 Manitoba                  2
```

```
      12 Ontario-South             4
      12 Quebec                    2
      12 -                         8
       - -                       155
```

Example C6-3:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS(ROLLUP(WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) ),
                       ROLLUP( MONTH(SALES_DATE), REGION ) )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

This results in:

```
WEEK     DAY_WEEK   MONTH   REGION        UNITS_SOLD
-------- ---------- ------- ------------- -----------
      13          6       - -                    27
      13          7       - -                    46
      13          -       - -                    73
      14          1       - -                    31
      14          2       - -                    43
      14          -       - -                    74
      53          1       - -                     8
      53          -       - -                     8
       -          -       3 Manitoba             22
       -          -       3 Ontario-North         8
       -          -       3 Ontario-South        34
       -          -       3 Quebec               40
       -          -       3 -                   104
       -          -       4 Manitoba             17
       -          -       4 Ontario-North         1
       -          -       4 Ontario-South        14
       -          -       4 Quebec               11
       -          -       4 -                    43
       -          -      12 Manitoba              2
       -          -      12 Ontario-South         4
       -          -      12 Quebec                2
       -          -      12 -                     8
       -          -       - -                   155
       -          -       - -                   155
```

Using the two ROLLUPs as grouping sets causes the result to include duplicate rows. There are even two grand total rows.

Observe how the use of ORDER BY has affected the results:

- In the first grouped set, week 53 has been repositioned to the end
- In the second grouped set, month 12 has now been positioned to the end and the regions now appear in alphabetic order.
- Null values are sorted high.

Example C7:

In queries that perform multiple ROLLUPs in a single pass (such as Example C6-3) you may want to be able to indicate which grouping set produced each row. The following steps demonstrate how to provide a column (called GROUP) which indicates the origin of each row in the result set. By origin means which one of the two grouping sets produced the row in the result set.

Step 1: Introduce a way of generating new data values using a query which selects from a VALUES clause (which is an alternate form of a fullselect). This query shows how a table called X can be derived that has 2 columns, R1 and R2, and one row of data.

```
SELECT R1, R2
FROM (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2)
```

Results in:

```
R1       R2
-------- -------
GROUP 1  GROUP 2
```

Step 2: Form the cross product of this table X with the SALES table. This adds columns R1 and R2 to every row.

```
SELECT R1, R2,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SALES AS UNITS_SOLD
FROM SALES,
     (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2)
```

Step 3: Now these columns are combined with the grouping sets to include these columns in the rollup analysis.

```
SELECT R1, R2,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES,
     (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2)
GROUP BY GROUPING SETS((R1, ROLLUP(WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE)) ),
                       (R2, ROLLUP(MONTH(SALES_DATE), REGION)) )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

This results in:

```
R1       R2      WEEK     DAY_WEEK   MONTH   REGION        UNITS_SOLD
-------- ------- -------- ---------- ------- ------------- ----------
GROUP 1  -             13          6       - -                     27
GROUP 1  -             13          7       - -                     46
GROUP 1  -             13          -       - -                     73
GROUP 1  -             14          1       - -                     31
GROUP 1  -             14          2       - -                     43
GROUP 1  -             14          -       - -                     74
GROUP 1  -             53          1       - -                      8
GROUP 1  -             53          -       - -                      8
-        GROUP 2        -          -       3 Manitoba              22
-        GROUP 2        -          -       3 Ontario-North          8
-        GROUP 2        -          -       3 Ontario-South         34
-        GROUP 2        -          -       3 Quebec                40
-        GROUP 2        -          -       3 -                    104
-        GROUP 2        -          -       4 Manitoba              17
-        GROUP 2        -          -       4 Ontario-North          1
-        GROUP 2        -          -       4 Ontario-South         14
-        GROUP 2        -          -       4 Quebec                11
-        GROUP 2        -          -       4 -                     43
-        GROUP 2        -          -      12 Manitoba               2
-        GROUP 2        -          -      12 Ontario-South          4
-        GROUP 2        -          -      12 Quebec                 2
```

```
-        GROUP 2        -          -      12 -                    8
-        GROUP 2        -          -       - -                  155
GROUP 1  -              -          -       - -                  155
```

Step 4: Notice that because R1 and R2 are used in different grouping sets, whenever R1 is non-null in the result, R2 is null and whenever R2 is non-null in the result, R1 is null. That means you can consolidate these columns into a single column using the COALESCE function. You can also use this column in the ORDER BY clause to keep the results of the two grouping sets together.

```
SELECT COALESCE(R1, R2) AS GROUP,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES,
         (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2)
GROUP BY GROUPING SETS((R1, ROLLUP(WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE)) ),
                       (R2, ROLLUP(MONTH(SALES_DATE), REGION)) )
ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION
```

This results in:

```
GROUP    WEEK     DAY_WEEK   MONTH   REGION        UNITS_SOLD
-------- -------- ---------- ------- ------------- -----------
GROUP 1        13          6       - -                    27
GROUP 1        13          7       - -                    46
GROUP 1        13          -       - -                    73
GROUP 1        14          1       - -                    31
GROUP 1        14          2       - -                    43
GROUP 1        14          -       - -                    74
GROUP 1        53          1       - -                     8
GROUP 1        53          -       - -                     8
GROUP 1         -          -       - -                   155
GROUP 2         -          -       3 Manitoba             22
GROUP 2         -          -       3 Ontario-North         8
GROUP 2         -          -       3 Ontario-South        34
GROUP 2         -          -       3 Quebec               40
GROUP 2         -          -       3 -                   104
GROUP 2         -          -       4 Manitoba             17
GROUP 2         -          -       4 Ontario-North         1
GROUP 2         -          -       4 Ontario-South        14
GROUP 2         -          -       4 Quebec               11
GROUP 2         -          -       4 -                    43
GROUP 2         -          -      12 Manitoba              2
GROUP 2         -          -      12 Ontario-South         4
GROUP 2         -          -      12 Quebec                2
GROUP 2         -          -      12 -                     8
GROUP 2         -          -       - -                   155
```

Example C8

The following example illustrates the use of various aggregate functions when performing a CUBE. The example also makes use of cast functions and rounding to produce a decimal result with reasonable precision and scale.

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS SALES,
       MAX(SALES) AS BEST_SALE,
       CAST(ROUND(AVG(DECIMAL(SALES)),2) AS DECIMAL(5,2)) AS AVG_UNITS_SOLD
FROM SALES
GROUP BY CUBE (MONTH(SALES_DATE), REGION)
ORDER BY MONTH, REGION
```

This results in:

```
MONTH   REGION        UNITS_SOLD   BEST_SALE  AVG_UNITS_SOLD
------- ------------- ------------ ---------- --------------
      3 Manitoba               22          7           3.14
      3 Ontario-North           8          3           2.67
      3 Ontario-South          34         14           4.25
      3 Quebec                 40         18           5.00
      3 -                     104         18           4.00
      4 Manitoba               17          9           5.67
      4 Ontario-North           1          1           1.00
      4 Ontario-South          14          8           4.67
      4 Quebec                 11          8           5.50
      4 -                      43          9           4.78
     12 Manitoba                2          2           2.00
     12 Ontario-South           4          3           2.00
     12 Quebec                  2          1           1.00
     12 -                       8          3           1.60
      - Manitoba               41          9           3.73
      - Ontario-North           9          3           2.25
      - Ontario-South          52         14           4.00
      - Quebec                 53         18           4.42
      - -                     155         18           3.87
```

having-clause

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the *search-condition* is true. R is the result of the previous clause of the subselect. If this clause is not GROUP BY, R is considered a single group with no grouping expressions.

Each expression that contains a *column-name* in the search condition must do one of the following:

- Unambiguously identify a grouping expression of R.
- Be specified within an aggregate function.
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a table, view, common table expression, or nested table expression identified in an outer subselect.

The RRN, RID, DATAPARTITIONNAME, DATAPARTITIONNUM, DBPARTITIONNAME, DBPARTITIONNUM, and HASHED_VALUE functions cannot be specified in the HAVING clause unless it is within an aggregate function. See "Aggregate functions" on page 237 for restrictions that apply to the use of aggregate functions.

A group of R to which the search condition is applied supplies the argument for each aggregate function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see examples 6 and 7 under "Examples of a subselect" on page 674.

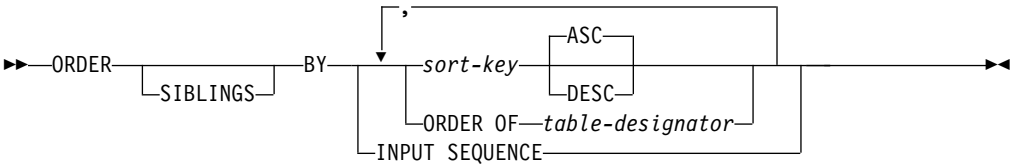
A correlated reference to a group of R must either identify a grouping column or be contained within an aggregate function.

When HAVING is used without GROUP BY, any column name in the select list must appear within an aggregate function.

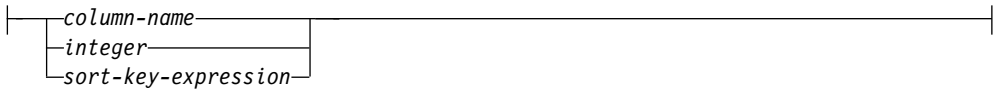
If a collating sequence other than *HEX is in effect when the statement that contains the HAVING clause is executed and if the *search-condition* contains operands that have SBCS data, mixed data, or Unicode data, the comparison for those predicates is done using weighted values. The weighted values are derived by applying the collating sequence to the operands in the predicate.

order-by-clause

The ORDER BY clause specifies an ordering of the rows of the result table.



sort-key:



A subselect that contains an ORDER BY clause cannot be specified in the outermost fullselect of a view.

Note: An ORDER BY clause in a subselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

If the subselect is not enclosed within parentheses and is not the outermost fullselect, the ORDER BY clause cannot be specified.

If a single sort specification (one *sort-key* with associated ascending or descending ordering specification) is identified, the rows are ordered by the values of that specification. If more than one sort specification is identified, the rows are ordered by the values of the first identified sort specification, then by the values of the second identified sort specification, and so on.

If the subselect includes a *hierarchical-query-clause*, ORDER SIBLINGS BY can be specified. This specifies that the ordering applies to siblings within the hierarchies only and parent rows are sorted before their child rows.

If a collating sequence other than *HEX is in effect when the statement that contains the ORDER BY clause is executed and if the ORDER BY clause involves sort specifications that are SBCS data, mixed data, or Unicode data, the comparison for those sort specifications is done using weighted values. The weighted values are derived by applying the collating sequence to the values of the sort specifications.

A named column in the select list may be identified by a *sort-key* that is a *integer* or a *column-name*. An unnamed column in the select list may be identified by a *integer* or, in some cases by a *sort-key-expression* that matches the expression in the select list (see details of *sort-key-expression*). “Names of result columns” on page 631 defines when result columns are unnamed. If the fullselect includes a UNION operator, see “fullselect” on page 676 for the rules on named columns in a fullselect.

Ordering is performed in accordance with the comparison rules described in Chapter 2, “Language elements,” on page 49. The null value is higher than all other values. If your ordering specification does not determine a complete

ordering, rows with duplicate values of the last identified *sort-key* have an arbitrary order. If the ORDER BY clause is not specified, the rows of the result table have an arbitrary order.

The sum of the length attributes of the *sort-keys* must not exceed 3.5 gigabytes.

column-name
Must unambiguously identify a column of the result table. The column must not be a DATALINK or XML column and must not be the result of the ARRAY_AGG function. The rules for unambiguous column references are the same as in the other clauses of the fullselect. See "Column name qualifiers to avoid ambiguity" on page 142 for more information.

If the fullselect includes a UNION, UNION ALL, EXCEPT, or INTERSECT, the column name cannot be qualified.

The *column-name* may also identify a column name of a table, view, or *nested-table-expression* identified in the FROM clause. This includes columns defined as implicitly hidden. An error occurs if the subselect includes an aggregation in the select list and the *column-name* is not a *grouping-expression*.

integer
Must be greater than 0 and not greater than the number of columns in the result table. The integer *n* identifies the *n*th column of the result table. The identified column must not be a DATALINK or XML column and must not be the result of the ARRAY_AGG function.

sort-key-expression
An expression that is not simply a column name or an unsigned integer constant. The query to which ordering is applied must be a subselect to use this form of *sort-key*.

The *sort-key-expression* cannot contain RRN, RID, DATAPARTITIONNAME, DATAPARTITIONNUM, DBPARTITIONNAME, DBPARTITIONNUM, and HASHED_VALUE scalar functions if the *fullselect* includes a UNION, UNION ALL, EXCEPT, or INTERSECT. The result of the *sort-key-expression* must not be DATALINK or XML.

If the subselect is grouped, the *sort-key-expression* can be an expression in the select list of the subselect or can include an aggregate function, constant, or variable.

ASC
Uses the values of the column in ascending order. This is the default.

DESC
Uses the values of the column in descending order.

ORDER OF *table-designator*
Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause and the table reference must identify a *nested-table-expression* or *common-table-expression*. The ordering that is applied is the same as if the columns of the ORDER BY clause in the *nested-table-expression* or *common-table-expression* were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause. If the *nested-table-expression* or *common-table-expression* has no ORDER BY clause, the order is not defined.

ORDER OF is not allowed if the query specifies:

- a distributed table,
- a table with a read trigger, or
- a logical file built over multiple physical file members.

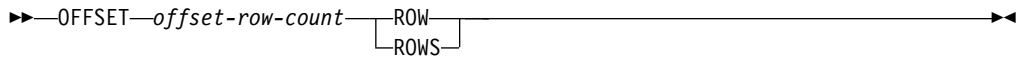
INPUT SEQUENCE
Specifies that the result table reflects the input order of the rows of an INSERT statement. INPUT SEQUENCE ordering can be specified only when an INSERT statement is specified in a *from-clause*. If INPUT SEQUENCE is specified and the input data is not ordered, the INPUT SEQUENCE clause is ignored.

A *sort-key* must not be LOB if the query specifies:

- a distributed table,
- a table with a read trigger, or
- a logical file built over multiple physical file members.

offset-clause

The *offset-clause* sets the number of rows to skip before any rows are retrieved. It lets the database manager know that the application does not want to start retrieving rows until *offset-row-count* rows have been skipped. If *offset-clause* is not specified, the default is equivalent to OFFSET 0 ROWS. An attempt to skip more rows than the number of rows in the intermediate result table is handled the same way as an empty result table.



offset-row-count

A numeric constant or numeric variable (except not a global variable) that specifies the number of rows to skip before any rows are retrieved. The numeric value is cast to BIGINT before processing. The value of *offset-row-count* must be positive or zero. It cannot be the null value.

Determining a predictable set of rows to skip requires the specification of an ORDER BY clause with sort keys that uniquely identify the sort order of each row in the intermediate result table. If the intermediate result table includes duplicate sort keys for some rows, the order of the rows is not deterministic. If there is no ORDER BY clause, the intermediate result table is not in a deterministic order. If the order of the intermediate result table is not deterministic, the set of skipped rows is unpredictable.

Notes

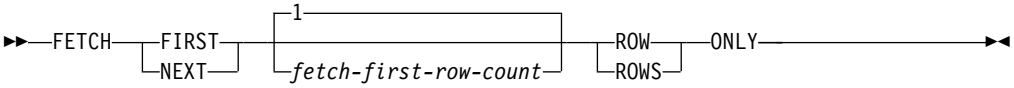
Allowed use of OFFSET: The *offset-clause* is only allowed as part of the outer fullselect of a DECLARE CURSOR statement or a prepared *select-statement*. It is not supported in interactive SQL.

select-list restriction: The *select-list* of the fullselect containing an *offset-clause* must not contain a NEXT VALUE sequence expression and must not contain a function that is non-deterministic, external action, or modifies SQL data.

Syntax alternatives: See the Notes entry associated with the *fetch-first-clause* for alternative syntax to set the number of rows to skip when specifying the maximum number of rows to retrieve.

fetch-first-clause

The *fetch-first-clause* sets a maximum number of rows that can be retrieved. It lets the database manager know that the application does not want to retrieve more than *fetch-first-row-count* rows, regardless of how many rows there are in the intermediate result table. An attempt to fetch beyond *fetch-first-row-count* rows is handled the same way as normal end of data.



fetch-first-row-count
A numeric constant or variable (except not a global variable) that specifies the maximum number of rows to retrieve. The numeric value is cast to BIGINT before processing. The value of *fetch-first-row-count* must be positive. It cannot be the null value.

Limiting the result table to a specified number of rows can improve performance. In some cases, the database manager will cease processing the query when it has determined the specified number of rows. If the *offset-clause* is also specified with a constant for *offset-row-count*, the database manager would also consider the constant offset value in determining when to cease processing.

Determining a predictable set of rows to retrieve requires the specification of an ORDER BY clause with sort keys that uniquely identify the sort order of each row in the intermediate result table. If the intermediate result table includes duplicate sort keys for some rows, the order of the rows is not deterministic. If there is no ORDER BY clause, the intermediate result table is not in a deterministic order. If the order of the intermediate result table is not deterministic, the set of rows retrieved is unpredictable.

If both the *order-by-clause* and *fetch-first-clause* are specified, the FETCH FIRST operation is always performed on the ordered data.

Notes

Allowed use of variable: *fetch-first-row-count* can be specified as a variable only as part of the outer fullselect of a DECLARE CURSOR statement or a prepared *select-statement*.

FIRST and NEXT: The keywords FIRST and NEXT can be used interchangeably. The result is unchanged; however, using the keyword NEXT is generally more readable when using the *offset-clause*.

Syntax alternatives: The following are supported for compatibility with SQL used by other database products. These alternatives are non-standard. The OFFSET clause and the LIMIT clause can only be specified in the outermost fullselect of a query.

Table 62.

Alternative syntax	Equivalent syntax
LIMIT *x*	FETCH FIRST *x* ROWS ONLY
LIMIT *x* OFFSET *y*	OFFSET *y* ROWS FETCH FIRST *x* ROWS ONLY

Table 62. (continued)

Alternative syntax	Equivalent syntax
LIMIT *y*, *x*	OFFSET *y* ROWS FETCH FIRST *x* ROWS ONLY

Examples of a subselect

subselect can be used in many different ways.

Example 1

Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

Example 2

Join the EMPPROJACT and EMPLOYEE tables, select all the columns from the EMPPROJACT table and add the employee's surname (LASTNAME) from the EMPLOYEE table to each row of the result.

```
SELECT EMPPROJACT.*, LASTNAME
  FROM EMPPROJACT, EMPLOYEE
  WHERE EMPPROJACT.EMPNO = EMPLOYEE.EMPNO
```

Example 3

Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO), employee surname (LASTNAME), department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the DEPARTMENT table) and department name (DEPTNAME) of all employees who were born (BIRTHDATE) earlier than 1930.

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
  FROM EMPLOYEE, DEPARTMENT
  WHERE WORKDEPT = DEPTNO
  AND YEAR(BIRTHDATE) < 1930
```

This subselect could also be written as follows:

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
  FROM EMPLOYEE INNER JOIN DEPARTMENT
  ON WORKDEPT = DEPTNO
  WHERE YEAR(BIRTHDATE) < 1930
```

Example 4

Select the job (JOB) and the minimum and maximum salaries (SALARY) for each group of rows with the same job code in the EMPLOYEE table, but only for groups with more than one row and with a maximum salary greater than or equal to 27000.

```
SELECT JOB, MIN(SALARY), MAX(SALARY)
  FROM EMPLOYEE
  GROUP BY JOB
  HAVING COUNT(*) > 1 AND MAX(SALARY) >= 27000
```

Example 5

Select all the rows of EMPPROJACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)

```
SELECT * FROM EMPPROJACT
  WHERE EMPNO IN (SELECT EMPNO FROM EMPLOYEE
                  WHERE WORKDEPT = 'E11')
```

Example 6

From the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary for all employees.

```
SELECT WORKDEPT, MAX(SALARY)
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                        FROM EMPLOYEE)
```

The subquery in the HAVING clause would only be executed once in this example.

Example 7

Using the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary in all other departments.

```
SELECT WORKDEPT, MAX(SALARY)
  FROM EMPLOYEE EMP_COR
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                        FROM EMPLOYEE
                        WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

In contrast to example 6, the subquery in the HAVING clause would need to be executed for each group.

Example 8

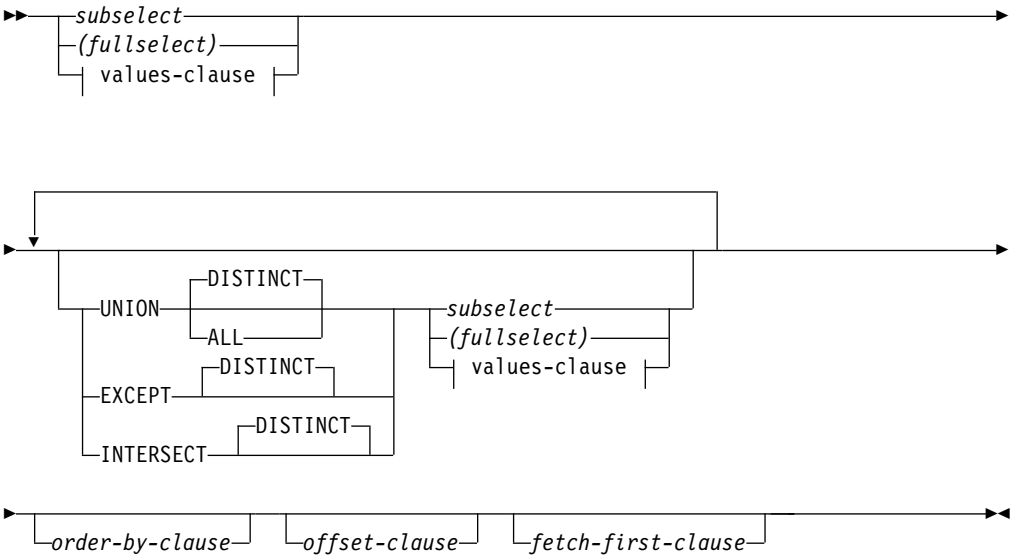
Join the EMPLOYEE and EMPPROJACT tables, select all of the employees and their project numbers. Return even those employees that do not have a project number currently assigned.

```
SELECT EMPLOYEE.EMPNO, PROJNO
  FROM EMPLOYEE LEFT OUTER JOIN EMPPROJACT
  ON EMPLOYEE.EMPNO = EMPPROJACT.EMPNO
```

Any employee in the EMPLOYEE table that does not have a project number in the EMPPROJACT table will return one row in the result table containing the EMPNO value and the null value in the PROJNO column.

fullselect

The *fullselect* is a component of the *select-statement*, ALTER TABLE statement for the definition of a materialized query table, CREATE TABLE statement, CREATE VIEW statement, DECLARE GLOBAL TEMPORARY TABLE statement, INSERT statement, SET transition-variable statement, SET VARIABLE statement, UPDATE statement, and *assignment-statement*.



values-clause:

values-row:

A fullselect that is enclosed in parenthesis is called a *subquery*. For example, a *subquery* can be used in a search condition.

A *scalar-fullselect* is a fullselect, enclosed in parentheses, that returns a single result row and a single result column. If the result of the fullselect is no rows, then the null value is returned. An error is returned if there is more than one row in the result.

A *fullselect* specifies a result table. If UNION, EXCEPT, or INTERSECT is not used, the result of the fullselect is the result of the specified subselect or *values-clause*.

values-clause

Derives a result table by specifying the actual values, using expressions, for each column of a row in the result table. Multiple rows may be specified.

NULL can only be used with multiple specifications of *values-row*, and at least one row in the same column must not be NULL.

A *values-row* is specified by:

- A single expression for a single column result table or,
- *n* expressions (or NULL) separated by commas and enclosed in parentheses, where *n* is the number of columns in the result table.

A multiple row VALUES clause must have the same number of expressions in each *values-row*.

The following are examples of *values-clauses* and their meanings.

```
VALUES (1), (2), (3)           - 3 rows of 1 column
VALUES 1, 2, 3                 - 3 rows of 1 column
VALUES (1, 2, 3)               - 1 row of 3 columns
VALUES (1,21), (2,22), (3,23)  - 3 rows of 2 columns
```

A *values-clause* that is composed of *n* specifications of *values-row*, RE_1 to RE_n, where *n* is greater than 1 is equivalent to:

RE_1 UNION ALL RE_2 ... UNION ALL RE_n

This means that the corresponding expressions of each *values-row* must be comparable. All result columns in a *values-row* are unnamed.

UNION, EXCEPT, or INTERSECT

The set operators UNION, EXCEPT, and INTERSECT correspond to the relational operators union, difference, and intersection. A fullselect specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect. Otherwise, the result table is derived by combining two other result tables (R1 and R2) subject to the specified set operator.

UNION DISTINCT or UNION ALL

If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

EXCEPT DISTINCT

The result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated. Each row in the result table of the difference is a row from R1 that does not have a matching row in R2.

INTERSECT DISTINCT

The result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated. Each row in the result table of the intersection is a row that exists in both R1 and R2.

Rules for columns:

- R1 and R2 must have the same number of columns, and the data type of the *n*th column of R1 must be compatible with the data type of the *n*th column of R2. Character-string values are compatible with datetime values.
- The *n*th column of the result of UNION, UNION ALL, EXCEPT, or INTERSECT is derived from the *n*th columns of R1 and R2. The attributes of the result columns are determined using the rules for result columns.

- If the *n*th column of R1 is named, then the *n*th column of the result table has that result column name. Otherwise, the result column is unnamed.
- If UNION, INTERSECT, or EXCEPT is specified, no column can be a DATALINK or XML column.

For information on the valid combinations of operand columns and the data type of the result column, see "Rules for result data types" on page 115.

EXCEPT and INTERSECT restrictions: VALUES, INTERSECT, and EXCEPT are not allowed if the query specifies one of the following:

- A distributed table
- A table with a read trigger
- A logical file built over multiple physical file members

Duplicate rows: Two rows are duplicates if each value in the first is equal to the corresponding value of the second. (For determining duplicates, two null values are considered equal.)

Operator precedence: UNION, UNION ALL, and INTERSECT are associative set operations. However, when UNION, UNION ALL, EXCEPT, and INTERSECT are used in the same statement, the result depends on the order in which the operations are performed. Operations within parenthesis are performed first. When the order is not specified by parentheses, operations are performed in left-to-right order with the exception that all INTERSECT operations are performed before UNION or EXCEPT operations.

Results of set operators: In the following example, the values of tables R1 and R2 are shown on the left. The other headings listed show the values as a result of various set operations on R1 and R2.

R1	R2	UNION ALL	UNION	EXCEPT	INTERSECT
1	1	1	1	2	1
1	1	1	2	5	3
1	3	1	3		4
2	3	1	4		
2	3	1	5		
2	3	2			
3	4	2			
4		2			
4		3			
5		3			
		3			
		3			
		3			
		4			
		4			
		4			
		5			

Collating sequence: If a collating sequence other than *HEX is in effect when the statement that contains the UNION, EXCEPT, or INTERSECT keyword is executed and if the result tables contain columns that are SBCS data, mixed data, or Unicode data, the comparison for those columns is done using weighted values. The weighted values are derived by applying the collating sequence to each value.

Examples of a fullselect

Example 1

Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

Example 2

List the employee numbers (EMPNO) of all employees in the EMPLOYEE table whose department number (WORKDEPT) either begins with 'E' **or** who are assigned to projects in the EMPPROJACT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

```
SELECT EMPNO FROM EMPLOYEE
  WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO FROM EMPPROJACT
  WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

Example 3

Make the same query as in example 2, only use UNION ALL so that no duplicate rows are eliminated.

```
SELECT EMPNO FROM EMPLOYEE
  WHERE WORKDEPT LIKE 'E%'
UNION ALL
SELECT EMPNO FROM EMPPROJACT
  WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

Example 4

Make the same query as in example 2, and, in addition, "tag" the rows from the EMPLOYEE table with 'emp' and the rows from the EMPPROJACT table with 'empproject'. Unlike the result from example 2, this query may return the same EMPNO more than once, identifying which table it came from by the associated "tag".

```
SELECT EMPNO, 'emp' FROM EMPLOYEE
  WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'empprojact' FROM EMPPROJACT
  WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

Example 5

This example of EXCEPT produces all rows that are in T1 but not in T2, with duplicate rows removed.

```
(SELECT * FROM T1)
     EXCEPT DISTINCT
(SELECT * FROM T2)
```

If no NULL values are involved, this example returns the same results as:

```
(SELECT DISTINCT *
   FROM T1
   WHERE NOT EXISTS (SELECT * FROM T2
                       WHERE  T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...) )
```

where C1, C2, and so on represent the columns of T1 and T2.

Example 6

This example of INTERSECT produces all rows that are in both tables T1 and T2, with duplicate rows removed.

```
(SELECT * FROM T1)
     INTERSECT DISTINCT
(SELECT * FROM T2)
```

If no NULL values are involved, this example returns the same results as:

```
(SELECT DISTINCT *
   FROM T1
   WHERE EXISTS (SELECT * FROM T2
                   WHERE  T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...) )
```

where C1, C2, and so on represent the columns of T1 and T2.

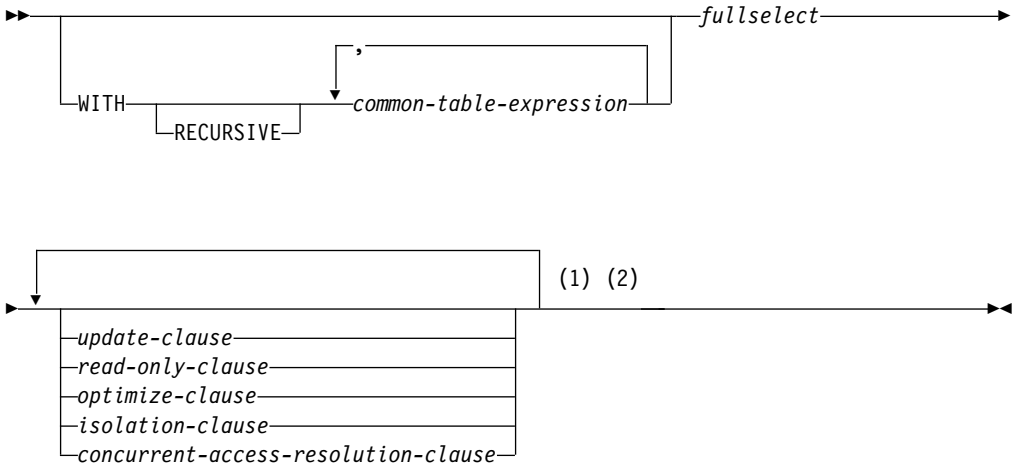
Example 7

Make the same query as in example 3, only include an additional two employees currently not in any table and tag these rows as "new".

```
SELECT EMPNO, 'emp' FROM EMPLOYEE
  WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'empprojact' FROM EMPPROJACT
  WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
UNION
VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new')
```

select-statement

The *select-statement* is the form of a query that can be directly specified in a DECLARE CURSOR or FOR statement, prepared and then referenced in a DECLARE CURSOR statement, or directly specified in an SQLJ assignment clause. It can also be issued interactively. In any case, the table specified by a *select-statement* is the result of the fullselect.



Notes:

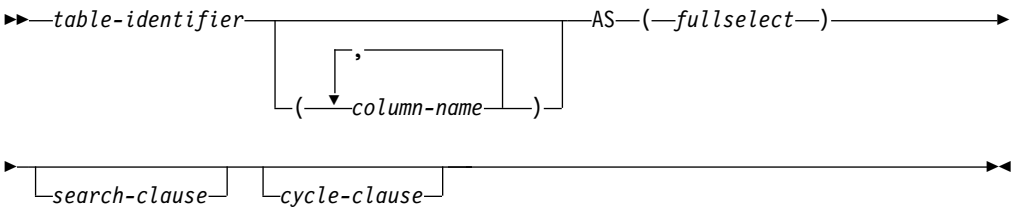
1 The update-clause and read-only-clause cannot both be specified in the same select-statement.

2 Each clause may be specified only once.

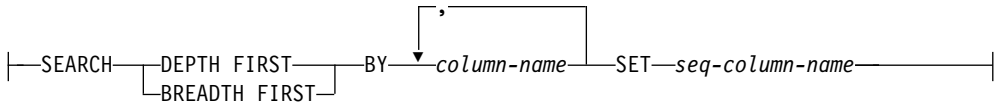
RECURSIVE
Indicates that a *common-table-expression* is potentially recursive.

common-table-expression

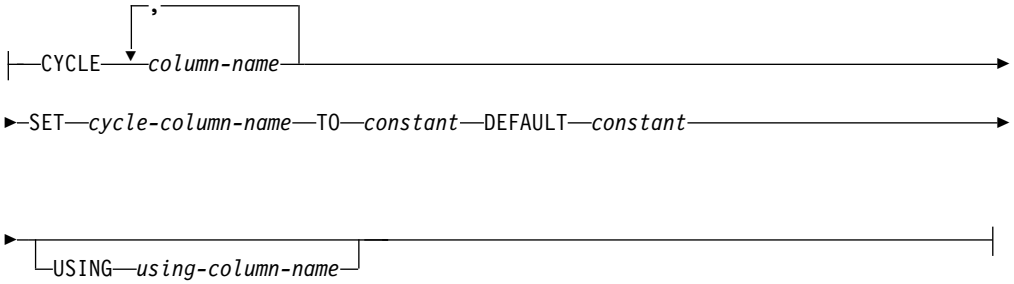
A *common-table-expression* permits defining a result table with a *table-identifier* that can be specified as a table name in any FROM clause of the fullselect that follows. Multiple common table expressions can be specified following the single WITH keyword. Each common table expression specified can also be referenced by name in the FROM clause of subsequent common table expressions.



search-clause:



cycle-clause:



If a list of columns is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If these column names are not specified, the names are derived from the select list of the subselect used to define the common table expression.

The *table-identifier* of a common table expression must be different from any other common table expression *table-identifier* in the same statement. A common table expression *table-identifier* can be specified as a table name in any FROM clause throughout the fullselect. A *table-identifier* of a common table expression overrides any existing table, view, or alias (in the catalog) with the same unqualified name or any *table-identifier* specified for a trigger.

If more than one common table expression is defined in the same statement, cyclic references between the common table expressions are not permitted. A *cyclic reference* occurs when two common table expressions *dt1* and *dt2* are created such that *dt1* refers to *dt2* and *dt2* refers to *dt1*.

The *table name* of a common table expression can only be referenced in the *select-statement*, INSERT statement, or CREATE VIEW statement that defines it.

If a *select-statement*, INSERT statement, or CREATE VIEW statement refers to an unqualified table name, the following rules are applied to determine which table is actually being referenced:

- If the unqualified name corresponds to one or more common table expression names that are specified in the *select-statement*, the name identifies the common table expression that is in the innermost scope.
- If in a CREATE TRIGGER statement and the unqualified name corresponds to a transition table name, the name identifies that transition table.
- Otherwise, the name identifies a persistent table, a temporary table, or a view that is present in the default schema.

A *common-table-expression* can be used:

- In place of a view to avoid creating the view (when general use of the view is not required and positioned UPDATE or DELETE is not used)
- To enable grouping by a column that is derived from a scalar-fullselect or function that is not deterministic
- When the required result table is based on variables
- When the same result table needs to be shared in a *fullselect*
- When the result needs to be derived using recursion

If a *fullselect* of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive table expression*. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The following restrictions must be true of a recursive *common-table-expression*:

- Each fullselect that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed.
- The UNION ALL set operator must be specified.
- A list of *column-names* must be specified following the *table-identifier* of the *common-table-expression*.
- The first *fullselect* of the first union (the initialization *fullselect*) must not include a reference to the *common-table-expression* itself in any FROM clause.
- Each *fullselect* that is part of the recursion cycle must not include any aggregate functions, GROUP BY clauses, or HAVING clauses.
- The FROM clauses of these *fullselects* can include at most one reference to a *common-table-expression* that is part of a recursion cycle.
- The table being defined in the *common-table-expression* cannot be referenced in a subquery of a *fullselect* that defines the *common-table-expression*.
- LEFT OUTER JOIN and FULL OUTER JOIN are not allowed if the *common-table-expression* is the right operand. RIGHT OUTER JOIN and FULL OUTER JOIN are not allowed if the *common-table-expression* is the left operand.
- Each *fullselect* other than the initialization *fullselect* that is part of the recursion cycle must not include an ORDER BY clause.

If a column name of the *common-table-expression* is referred to in the iterative *fullselect*, the attributes of the result columns are determined using the rules for result columns. For more information see “Rules for result data types” on page 115.

search-clause
The SEARCH clause in the definition of the recursive *common-table-expression* is used to specify the order in which the result rows are to be returned.

SEARCH DEPTH FIRST
Each parent or containing item appears in the result before the items that it contains.

SEARCH BREADTH FIRST
Sibling items are grouped before subordinate items.

BY *column-name*,...
Identifies the columns that associate the parent and child relationship of the recursive query. Each *column-name* must unambiguously identify a column of the parent. The column must not be a DATALINK or XML column. The rules for unambiguous column references are the same as in the other clauses of the fullselect. See "Column name qualifiers to avoid ambiguity" on page 142 for more information.

The *column-name* must identify a column name of the recursive *common-table-expression*. The *column-name* must not be qualified.

SET *seq-column-name*
Specifies the name of a result column that contains an ordinal number of the current row in the recursive query result. The data type of the *seq-column-name* is BIGINT.

The *seq-column-name* may only be referenced in the ORDER BY clause of the outer fullselect that references the *common-table-expression*. The *seq-column-name* cannot be referenced in the fullselect that defines the *common-table-expression*.

The *seq-column-name* must not be the same as *using-column-name* or *cycle-column-name*.

cycle-clause
The CYCLE clause in the definition of the recursive *common-table-expression* is used to prevent an infinite loop in the recursive query when the parent and child relationship of the data results in a loop.

CYCLE *column-name*,...
Specifies the list of columns that represent the parent/child join relationship values for the recursion. Any new row from the query is first checked for a duplicate value (per these column names) in the existing rows that lead to this row in the recursive query results to determine if there is a cycle.

Each *column-name* must identify a result column of the common table expression. It must not be an XML or DataLink column. The same *column-name* must not be specified more than once.

SET *cycle-column-name*
Specifies the name of a result column that is set based on whether a cycle has been detected in the recursive query:

- If a duplicate row is encountered, indicating that a cycle has been detected in the data, the *cycle-column-name* is set to the TO *constant*.
- If a duplicate row is not encountered, indicating that a cycle has not been detected in the data, the *cycle-column-name* is set to the DEFAULT *constant*.

The data type of the *cycle-column-name* is CHAR(1).

When cyclic data in the row is encountered, the duplicate row is not returned to the recursive query process for further recursion and that child branch of the query is stopped. By specifying the provided *cycle-column-name* is in the result set of the main fullselect, the existence of cyclic data can actually be determined and even corrected if that is wanted.

The *cycle-column-name* must not be the same as *using-column-name* or *seq-column-name*.

The *cycle-column-name* can be referenced in the fullselect that defines the *common-table-expression*.

TO *constant*
Specifies a CHAR(1) constant value to assign to the *cycle-column* if a cycle has been detected in the data. The TO *constant* must not be equal to the DEFAULT *constant*.

DEFAULT *constant*
Specifies a CHAR(1) constant value to assign to the *cycle-column* if a cycle has not been detected in the data. The DEFAULT *constant* must not be equal to the TO *constant*.

USING *using-column-name*
Identifies the temporary results consisting of the columns from the CYCLE column list. The temporary result is used by the database manager to identify duplicate rows in the query result.

The *using-column-name* must not be the same as *cycle-column-name* or *seq-column-name*.

When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Ensure that recursion cycles will end. This is especially important if the data involved is cyclic. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. The recursive common table expression is expected to include:

- In the iterative *fullselect*, an integer column incremented by a constant.
- A predicate in the WHERE clause of the iterative *fullselect* in the form "counter_col < constant" or "counter_col < :hostvar".

A warning is issued if this syntax is not found in the recursive common table expression.

Recursive common table expressions are not allowed if the query specifies:

- a distributed table,
- a table with a read trigger, or
- a logical file built over multiple physical file members.

Recursion example: bill of materials

Bill of materials (BOM) applications are a common requirement in many business environments. To illustrate the capability of a recursive common table expression for BOM applications, consider a table of parts with associated subparts and the quantity of subparts required by the part.

For this example, create the table as follows:

```
CREATE TABLE  PARTLIST
  ( PART     VARCHAR(8),
    SUBPART  VARCHAR(8),
    QUANTITY INTEGER )
```

To give query results for this example, assume that the PARTLIST table is populated with the following values:

```
PART     SUBPART  QUANTITY
-------- -------- -----------
   00       01         5
   00       05         3
   01       02         2
   01       03         3
   01       04         4
   01       06         3
   02       05         7
   02       06         6
   03       07         6
   04       08        10
   04       09        11
   05       10        10
   05       11        10
   06       12        10
   06       13        10
   07       14         8
   07       12         8
```

Example 1: Single level explosion

The first example is called single level explosion. It answers the question, "What parts are needed to build the part identified by '01'?". The list will include the direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
     (  SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
           FROM PARTLIST ROOT
           WHERE ROOT.PART = '01'
       UNION ALL
        SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
           FROM RPL PARENT, PARTLIST CHILD
           WHERE PARENT.SUBPART = CHILD.PART
      )
SELECT DISTINCT PART, SUBPART, QUANTITY
  FROM RPL
  ORDER BY PART, SUBPART, QUANTITY
```

The above query includes a common table expression, identified by the name RPL, that expresses the recursive part of this query. It illustrates the basic elements of a recursive common table expression.

The first operand (fullselect) of the UNION, referred to as the initialization fullselect, gets the direct children of part '01'. The FROM clause of this fullselect refers to the source table and will never refer to itself (RPL in this case). The result of this first fullselect goes into the common table expression RPL (Recursive PARTLIST). As in this example, the UNION must always be a UNION ALL.

The second operand (fullselect) of the UNION uses RPL to compute subparts of subparts by having the FROM clause refer to the common table expression RPL and the source table with a join of a part from the source table (child) to a subpart of the current result contained in RPL (parent). The result goes back to RPL again. The second operand of UNION is then used repeatedly until no more children exist.

The SELECT DISTINCT in the main fullselect of this query ensures the same part/subpart is not listed more than once.

The result of the query is as follows:

```
PART     SUBPART  QUANTITY
-------- -------- -----------
   01       02         2
   01       03         3
   01       04         4
   01       06         3
   02       05         7
   02       06         6
   03       07         6
   04       08        10
   04       09        11
   05       10        10
   05       11        10
   06       12        10
   06       13        10
   07       12         8
   07       14         8
```

Observe in the result that part '01' goes to '02' which goes to '06' and so on. Further, notice that part '06' is reached twice, once through '01' directly and another time through '02'. In the output, however, its subcomponents are listed only once (this is the result of using a SELECT DISTINCT) as required.

Example 2: Summarized explosion

The second example is a summarized explosion. The question posed here is, what is the total quantity of each part required to build part '01'. The main difference from the single level explosion is the need to aggregate the quantities. The first example indicates the quantity of subparts required for the part whenever it is required. It does not indicate how many of the subparts are needed to build part '01'.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
     (  SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
          FROM PARTLIST ROOT
          WHERE ROOT.PART = '01'
       UNION ALL
        SELECT PARENT.PART, CHILD.SUBPART, PARENT.QUANTITY*CHILD.QUANTITY
          FROM RPL PARENT, PARTLIST CHILD
          WHERE PARENT.SUBPART = CHILD.PART
      )
SELECT PART, SUBPART, SUM(QUANTITY) AS "Total QTY Used"
  FROM RPL
  GROUP BY PART, SUBPART
  ORDER BY PART, SUBPART
```

In the above query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name RPL, shows the aggregation of the quantity. To find out how much of a subpart is used, the quantity of the parent is multiplied by the quantity per parent of a child. If a part is used multiple times in different places, it requires another final aggregation. This is done by the grouping over the common table expression RPL and using the SUM aggregate function in the select list of the main fullselect.

The result of the query is as follows:

```
PART     SUBPART  Total Qty Used
-------- -------- --------------
   01       02          2
   01       03          3
   01       04          4
   01       05         14
```

```
01       06         15
01       07         18
01       08         40
01       09         44
01       10        140
01       11        140
01       12        294
01       13        150
01       14        144
```

Looking at the output, consider the line for subpart '06'. The total quantity used value of 15 is derived from a quantity of 3 directly for part '01' and a quantity of 6 for part '02' which is needed 2 times by part '01'.

Example 3: Controlling depth

The question may come to mind, what happens when there are more levels of parts in the table than you are interested in for your query? That is, how is a query written to answer the question, "What are the first two levels of parts needed to build the part identified by '01'?" For the sake of clarity in the example, the level is included in the result.

```
WITH RPL ( LEVEL, PART, SUBPART, QUANTITY)
  AS ( SELECT 1, ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
          FROM PARTLIST ROOT
          WHERE ROOT.PART = '01'
        UNION ALL
       SELECT PARENT.LEVEL+1, CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
          FROM RPL PARENT, PARTLIST CHILD
          WHERE PARENT.SUBPART = CHILD.PART
          AND PARENT.LEVEL < 2
     )
SELECT PART, LEVEL, SUBPART, QUANTITY
  FROM RPL
```

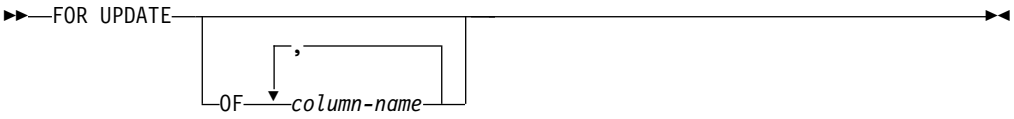
This query is similar to example 1. The column LEVEL was introduced to count the levels from the original part. In the initialization fullselect, the value for the LEVEL column is initialized to 1. In the subsequent fullselect, the level from the parent is incremented by 1. Then to control the number of levels in the result, the second fullselect includes the condition that the parent level must be less than 2. This ensures that the second fullselect only processes children to the second level.

The result of the query is:

```
PART     LEVEL       SUBPART  QUANTITY
-------- ----------- -------- -----------
   01         1         02         2
   01         1         03         3
   01         1         04         4
   01         1         06         3
   02         2         05         7
   02         2         06         6
   03         2         07         6
   04         2         08        10
   04         2         09        11
   06         2         12        10
   06         2         13        10
```

update-clause

The UPDATE clause identifies the columns that can appear as targets in an assignment clause in a subsequent positioned UPDATE statement. Each *column-name* must be unqualified and must identify a column of the table or view identified in the first FROM clause of the fullselect. The clause must not be specified if the result table of the fullselect is read-only.



If an UPDATE clause is specified with a *column-name* list, and extended indicator variables are not enabled, then *column-name* must be an updatable column.

If the UPDATE clause is specified without *column-name* list, then the implicit *column-name* list is determined as follows:

- If extended indicator variables are enabled, all the columns of the table or view identified in the first FROM clause of the fullselect.
- Otherwise, all the updatable columns of the table or view identified in the first FROM clause of the fullselect.

If the UPDATE clause is not specified in a *select-statement* and its result table is not *read-only*, then an implicit UPDATE clause will result. The implicit *column-name* list is determined as follows:

- If extended indicator variables are enabled, all the columns of the table or view identified in the first FROM clause of the fullselect are included.
- Otherwise, all the updatable columns of the table or view identified in the first FROM clause of the fullselect are included.

The UPDATE clause must not be specified if the result table of the fullselect is read-only (for more information see “DECLARE CURSOR” on page 1079) or if the FOR READ ONLY clause is used.

When the UPDATE clause is used, FETCH operations referencing the cursor acquire an exclusive row lock.

read-only-clause

The READ ONLY clause indicates that the result table is read-only; therefore, the cursor cannot be used for positioned UPDATE and DELETE statements.

Some result tables are read-only by nature. (For example, a table based on a read-only view). FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For result tables in which updates and deletes are allowed, specifying FOR READ ONLY can possibly improve the performance of FETCH operations by allowing the database manager to do blocking and avoid exclusive locks. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, the database manager might open cursors as if the UPDATE clause was specified.

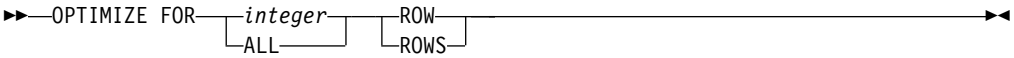
A read-only result table must not be referred to in an UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY.

To guarantee that selected data is not locked by any other job, you can specify the optional syntax of USE AND KEEP EXCLUSIVE LOCKS on the *isolation-clause*. This guarantees that the selected data can later be updated or deleted without incurring a row lock conflict.

Syntax Alternatives: FOR FETCH ONLY can be specified in place of FOR READ ONLY.

optimize-clause

The *optimize-clause* tells the database manager to assume that the program does not intend to retrieve more than *integer* rows from the result table. Without this clause, or with the keyword ALL, the database manager assumes that all rows of the result table are to be retrieved. Optimizing for *integer* rows can improve performance. The database manager will optimize the query based on the specified number of rows.

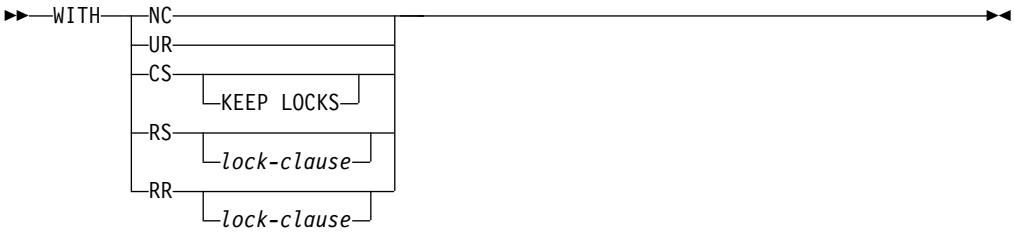


The clause does not change the result table or the order in which the rows are fetched. Any number of rows can be fetched, but performance can possibly degrade after *integer* fetches.

The value of *integer* must be a positive integer (not zero).

isolation-clause

The *isolation-clause* specifies an isolation level at which the select statement is executed.



lock-clause:

RR Repeatable Read

USE AND KEEP EXCLUSIVE LOCKS
Exclusive row locks are acquired and held until a COMMIT or ROLLBACK statement is executed.

RS Read Stability

USE AND KEEP EXCLUSIVE LOCKS
Exclusive row locks are acquired and held until a COMMIT or ROLLBACK statement is executed. The USE AND KEEP EXCLUSIVE LOCKS clause is only allowed in the *isolation-clause* in the following SQL statements:

- DECLARE CURSOR
- FOR
- *select-statement*
- SELECT INTO
- PREPARE in the ATTRIBUTES string

It is not allowed on updatable cursors.

CS Cursor Stability

KEEP LOCKS
The KEEP LOCKS clause specifies that any read locks acquired will be held for a longer duration. Normally, read locks are released when the next row is read. If the isolation clause is associated with a cursor, the locks will be held until the cursor is closed or until a COMMIT or ROLLBACK statement is executed. Otherwise, the locks will be held until the completion of the SQL statement.

UR Uncommitted Read

NC No Commit

If *isolation-clause* is not specified, the default isolation is used with the exception of a default isolation level of uncommitted read. See "Isolation level" on page 26 for a description of how the default is determined.

Exclusive locks: The USE AND KEEP EXCLUSIVE LOCKS clause should be used with caution. If it is specified, the exclusive row locks that are acquired on rows will prevent concurrent access to those rows by other users running COMMIT(*CS), COMMIT(*RS), and COMMIT(*RR) till the end of the unit of work. Concurrent access by users running COMMIT(*NC) or COMMIT(*UR) is not prevented.

Keyword Synonyms: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword NONE can be used as a synonym for NC.
- The keyword CHG can be used as a synonym for UR.
- The keyword ALL can be used as a synonym for RS.

concurrent-access-resolution-clause

The *concurrent-access-resolution-clause* specifies the concurrent access resolution to use for the statement.

SKIP LOCKED DATA
Specifies that the *select-statement*, searched UPDATE statement (including a searched update operation in a MERGE statement), or searched DELETE statement that is prepared in the PREPARE statement will skip rows when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. SKIP LOCKED DATA can be used only when isolation level NC, UR, CS, or RS is in effect.

SKIP LOCKED DATA is ignored if it is specified when the isolation level that is in effect is RR.

USE CURRENTLY COMMITTED
Specifies that the database manager can use the currently committed version of the data for applicable scans when it is in the process of being updated or deleted. Rows in the process of being inserted can be skipped. This clause applies if possible when the isolation level is CS without the KEEP LOCKS clause and is ignored otherwise. Applicable scans include read-only scans.

WAIT FOR OUTCOME
Specifies to wait for the commit or rollback when encountering data in the process of being updated or deleted. Rows encountered that are in the process of being inserted are not skipped. This clause applies if possible when the isolation level is CS or RS. It is ignored when an isolation level of NC, UR, or RR is in effect.

Examples of a select-statement

You can use select-statement in many different ways.

Example 1

Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

Example 2

Select the project name (PROJNAME), start date (PRSTDATE), and end date (PRENDATE) from the PROJECT table. Order the result table by the end date with the most recent dates appearing first.

```
SELECT PROJNAME, PRSTDATE, PRENDATE
  FROM PROJECT
  ORDER BY PRENDATE DESC
```

Example 3

Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the EMPLOYEE table. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY AVGSAL
```

Example 4

Declare a cursor named UP_CUR, to be used in a C program, that updates the start date (PRSTDATE) and the end date (PRENDATE) columns in the PROJECT table. The program must receive both of these values together with the project number (PROJNO) value for each row. The declaration specifies that the access path for the query be optimized for the retrieval of a maximum of 2 rows. Even so, the program can retrieve more than 2 rows from the result table. However, when more than 2 rows are retrieved, performance could possibly degrade.

```
EXEC SQL  DECLARE UP_CUR CURSOR FOR
            SELECT PROJNO, PRSTDATE, PRENDATE
              FROM PROJECT
              FOR UPDATE OF PRSTDATE, PRENDATE
              OPTIMIZE FOR 2 ROWS ;
```

Example 5

Select items from a table with an isolation level of Read Stability (RS).

```
SELECT NAME, SALARY
  FROM PAYROLL
  WHERE DEPT = 704
  WITH RS
```

Example 6

This example names the expression SALARY+BONUS+COMM as TOTAL_PAY:

```
SELECT SALARY+BONUS+COMM AS TOTAL_PAY
  FROM EMPLOYEE
  ORDER BY TOTAL_PAY
```

Example 7

Determine the employee number and salary of sales representatives along with the average salary and head count of their departments. Also, list the average salary of the department with the highest average salary.

Using a common table expression for this case saves the overhead of creating the DINFO view as a regular view. Because of the context of the rest of the fullselect, only the rows for the department of the sales representatives need to be considered by the view.

```
WITH
   DINFO (DEPTNO, AVGSALARY, EMPCOUNT) AS
     (SELECT OTHERS.WORKDEPT, AVG(OTHERS.SALARY), COUNT(*)
        FROM EMPLOYEE OTHERS
        GROUP BY OTHERS.WORKDEPT),
   DINFOMAX AS
     (SELECT MAX(AVGSALARY) AS AVGMAX
        FROM DINFO)
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT,
                          DINFOMAX.AVGMAX
  FROM EMPLOYEE THIS_EMP, DINFO, DINFOMAX
  WHERE THIS_EMP.JOB = 'SALESREP'
  AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

Example 8

Find the average charges for each subscriber (SNO) in the state of California during the last Friday of each month in the first quarter of 2000. Group the result according to SNO. Each MONTHnn table has columns for SNO, CHARGES, and DATE. The CUST table has columns for SNO and STATE.

```
SELECT V.SNO, AVG( V.CHARGES)
  FROM CUST, LATERAL (
    SELECT SNO, CHARGES, DATE
    FROM MONTH1
    WHERE DATE BETWEEN '01/01/2000' AND '01/31/2000'
        UNION ALL
    SELECT SNO, CHARGES, DATE
    FROM MONTH2
    WHERE DATE BETWEEN '02/01/2000' AND '02/29/2000'
        UNION ALL
    SELECT SNO, CHARGES, DATE
    FROM MONTH3
    WHERE DATE BETWEEN '03/01/2000' AND '03/31/2000'
    ) AS  V (SNO, CHARGES, DATE)
  WHERE CUST.SNO=V.SNO
  AND CUST.STATE='CA'
  AND DATE IN ('01/28/2000','02/25/2000','03/31/2000')
  GROUP BY V.SNO
```

Examples of a select-statement

Chapter 6. Statements

This section contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements.

The statements are listed in the following table.

Table 63. SQL Schema Statements

SQL Statement	Description	Page
ALTER FUNCTION (External Scalar)	Alters the description of an external scalar function	"ALTER FUNCTION (External Scalar)" on page 714
ALTER FUNCTION (External Table)	Alters the description of an external table function	"ALTER FUNCTION (External Table)" on page 719
ALTER FUNCTION (SQL Scalar)	Alters the description of an SQL scalar function	"ALTER FUNCTION (SQL Scalar)" on page 724
ALTER FUNCTION (SQL Table)	Alters the description of an SQL table function	"ALTER FUNCTION (SQL Table)" on page 731
ALTER PROCEDURE (External)	Alters the description of an external procedure	"ALTER PROCEDURE (External)" on page 739
ALTER PROCEDURE (SQL)	Alters the description of an SQL procedure	"ALTER PROCEDURE (SQL)" on page 744
ALTER SEQUENCE	Alters the description of a sequence	"ALTER SEQUENCE" on page 754
ALTER TABLE	Alters the description of a table	"ALTER TABLE" on page 759
COMMENT	Adds or replaces a comment to the description of an object	"COMMENT" on page 814
CREATE ALIAS	Creates an alias	"CREATE ALIAS" on page 847
CREATE FUNCTION	Creates a user-defined function	"CREATE FUNCTION" on page 851
CREATE FUNCTION (External Scalar)	Creates an external scalar function	"CREATE FUNCTION (External Scalar)" on page 856
CREATE FUNCTION (External Table)	Creates an external table function	"CREATE FUNCTION (External Table)" on page 876
CREATE FUNCTION (Sourced)	Creates a user-defined function based on another existing scalar or column function	"CREATE FUNCTION (Sourced)" on page 894
CREATE FUNCTION (SQL Scalar)	Creates an SQL scalar function	"CREATE FUNCTION (SQL Scalar)" on page 905
CREATE FUNCTION (SQL Table)	Creates an SQL table function	"CREATE FUNCTION (SQL Table)" on page 917
CREATE INDEX	Creates an index on a table	"CREATE INDEX" on page 929

Table 63. SQL Schema Statements (continued)

SQL Statement	Description	Page
CREATE PROCEDURE	Creates a procedure.	"CREATE PROCEDURE" on page 937
CREATE PROCEDURE (External)	Creates an external procedure.	"CREATE PROCEDURE (External)" on page 939
CREATE PROCEDURE (SQL)	Creates an SQL procedure.	"CREATE PROCEDURE (SQL)" on page 955
CREATE SCHEMA	Creates a schema and a set of objects in that schema	"CREATE SCHEMA" on page 968
CREATE SEQUENCE	Creates a sequence	"CREATE SEQUENCE" on page 974
CREATE TABLE	Creates a table	"CREATE TABLE" on page 982
CREATE TRIGGER	Creates a trigger	"CREATE TRIGGER" on page 1033
CREATE TYPE	Creates a type	"CREATE TYPE (Distinct)" on page 1055
CREATE VARIABLE	Creates a global variable	"CREATE VARIABLE" on page 1063
CREATE VIEW	Creates a view of one or more tables or views	"CREATE VIEW" on page 1069
DROP	Drops an alias, function, index, package, procedure, schema, sequence, table, trigger, type, variable, view, or XSR object.	"DROP" on page 1151
GRANT (Function or Procedure Privileges)	Grants privileges on a function or procedure	"GRANT (Function or Procedure Privileges)" on page 1219
GRANT (Global Variable Privileges)	Grants privileges on a global variable	"GRANT (Global Variable Privileges)" on page 1227
GRANT (Package Privileges)	Grants privileges on a package	"GRANT (Package Privileges)" on page 1230
GRANT (Sequence Privileges)	Grants privileges on a sequence	"GRANT (Sequence Privileges)" on page 1233
GRANT (Table or View Privileges)	Grants privileges on a table or view	"GRANT (Table or View Privileges)" on page 1236
GRANT (Type Privileges)	Grants privileges on a type	"GRANT (Type Privileges)" on page 1242
GRANT (XML Schema Privileges)	Grants privileges on an XML schema	"GRANT (XML Schema Privileges)" on page 1245
LABEL	Adds or replaces a label to the description of an object	"LABEL" on page 1263
RENAME	Renames a table, view, or index.	"RENAME" on page 1315
REVOKE (Function or Procedure Privileges)	Revokes privileges on a function or procedure	"REVOKE (Function or Procedure Privileges)" on page 1318
REVOKE (Global Variable Privileges)	Revokes privileges on a global variable	"REVOKE (Global Variable Privileges)" on page 1325

Table 63. SQL Schema Statements (continued)

SQL Statement	Description	Page
REVOKE (Package Privileges)	Revokes the privilege to execute statements in a package	"REVOKE (Package Privileges)" on page 1327
REVOKE (Sequence Privileges)	Revokes privileges on a sequence	"REVOKE (Sequence Privileges)" on page 1329
REVOKE (Table or View Privileges)	Revokes privileges on a table or view	"REVOKE (Table or View Privileges)" on page 1331
REVOKE (Type Privileges)	Revokes the privilege to use a type	"REVOKE (Type Privileges)" on page 1334
REVOKE (XML Schema Privileges)	Revokes privileges on an XML schema	"REVOKE (XML Schema Privileges)" on page 1336

Table 64. SQL Data Change Statements

SQL Statement	Description	Page
DELETE	Deletes one or more rows from a table	"DELETE" on page 1121
INSERT	Inserts one or more rows into a table	"INSERT" on page 1252
MERGE	Updates a target table using data from a source table	"MERGE" on page 1274
UPDATE	Updates the values of one or more columns in one or more rows of a table	"UPDATE" on page 1411

Table 65. SQL Data Statements

SQL Statement	Description	Page
	All SQL Data Change statements	Table 64
ALLOCATE CURSOR	Allocates a cursor for the result set identified by the result set locator variable.	"ALLOCATE CURSOR" on page 711
ASSOCIATE LOCATORS	Gets the result set locator value for each result set returned by a procedure.	"ASSOCIATE LOCATORS" on page 795
CLOSE	Closes a cursor	"CLOSE" on page 812
DECLARE CURSOR	Defines an SQL cursor	"DECLARE CURSOR" on page 1079
FETCH	Positions a cursor on a row of the result table; can also assign values from one or more rows of the result table to variables	"FETCH" on page 1173
FREE LOCATOR	Removes the association between a LOB locator variable and its value	"FREE LOCATOR" on page 1181
HOLD LOCATOR	Allows a LOB locator variable to retain its association with a value beyond a unit of work	"HOLD LOCATOR" on page 1248
LOCK TABLE	Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table	"LOCK TABLE" on page 1272
OPEN	Opens a cursor	"OPEN" on page 1287

Table 65. SQL Data Statements (continued)

SQL Statement	Description	Page
REFRESH TABLE	Refreshes the data in a materialized query table	"REFRESH TABLE" on page 1310
SELECT	Executes a query	"SELECT" on page 1344
SELECT INTO	Assigns values to variables	"SELECT INTO" on page 1345
SET transition-variable	Assigns values to a transition variable	"SET transition-variable" on page 1402
SET variable	Assigns values to a variable	"SET variable" on page 1404
VALUES	Provides a method to invoke a user-defined function from a trigger.	"VALUES" on page 1420
VALUES INTO	Specifies a result table of no more than one row and assigns the values to variables.	"VALUES INTO" on page 1422

Table 66. SQL Transaction Statements

SQL Statement	Description	Page
COMMIT	Ends a unit of work and commits the database changes made by that unit of work	"COMMIT" on page 824
RELEASE SAVEPOINT	Releases a savepoint within a unit of work	"RELEASE SAVEPOINT" on page 1314
ROLLBACK	Ends a unit of work and backs out the database changes made by that unit of work or made since the specified savepoint	"ROLLBACK" on page 1338
SAVEPOINT	Sets a savepoint within a unit of work	"SAVEPOINT" on page 1342
SET TRANSACTION	Changes the isolation level for the current unit of work	"SET TRANSACTION" on page 1399

Table 67. SQL Connection Statements

SQL Statement	Description	Page
CONNECT (Type 1)	Connects to an application server and establishes the rules for remote unit of work	"CONNECT (Type 1)" on page 836
CONNECT (Type 2)	Connects to an application server and establishes the rules for application-directed distributed unit of work	"CONNECT (Type 2)" on page 842
DISCONNECT	Immediately ends one or more connections	"DISCONNECT" on page 1149
RELEASE (Connection)	Places one or more connections in the release-pending state	"RELEASE (Connection)" on page 1312
SET CONNECTION	Establishes the application server of the process by identifying one of its existing connections	"SET CONNECTION" on page 1348

Table 68. SQL Dynamic Statements

SQL Statement	Description	Page
ALLOCATE DESCRIPTOR	Allocates an SQL descriptor	"ALLOCATE DESCRIPTOR" on page 712
Compound (dynamic)	Groups other statements together in an executable routine	"compound (dynamic)" on page 827
DEALLOCATE DESCRIPTOR	Deallocates an SQL descriptor	"DEALLOCATE DESCRIPTOR" on page 1078
DESCRIBE	Describes the result columns of a prepared statement	"DESCRIBE" on page 1127
DESCRIBE CURSOR	Describes the cursor.	"DESCRIBE CURSOR" on page 1132
DESCRIBE INPUT	Describes the input parameter markers of a prepared statement	"DESCRIBE INPUT" on page 1135
DESCRIBE PROCEDURE	Describes the result sets returned by a procedure.	"DESCRIBE PROCEDURE" on page 1139
DESCRIBE TABLE	Describes the columns of a table or view	"DESCRIBE TABLE" on page 1145
EXECUTE	Executes a prepared SQL statement	"EXECUTE" on page 1166
EXECUTE IMMEDIATE	Prepares and executes an SQL statement	"EXECUTE IMMEDIATE" on page 1170
GET DESCRIPTOR	Gets information from an SQL descriptor	"GET DESCRIPTOR" on page 1182
PREPARE	Prepares an SQL statement for execution	"PREPARE" on page 1293
SET DESCRIPTOR	Sets items in an SQL descriptor	"SET DESCRIPTOR" on page 1361

Table 69. SQL Session Statements

SQL Statement	Description	Page
DECLARE GLOBAL TEMPORARY TABLE	Defines a declared temporary table	"DECLARE GLOBAL TEMPORARY TABLE" on page 1089
SET CURRENT DEBUG MODE	Assigns a value to the CURRENT DEBUG MODE special register	"SET CURRENT DEBUG MODE" on page 1351
SET CURRENT DECFLOAT ROUNDING MODE	Assigns a value to the CURRENT DECFLOAT ROUNDING MODE special register	"SET CURRENT DECFLOAT ROUNDING MODE" on page 1353
SET CURRENT DEGREE	Assigns a value to the CURRENT DEGREE special register	"SET CURRENT DEGREE" on page 1356
SET CURRENT IMPLICIT XMLPARSE OPTION	Assigns a value to the CURRENT IMPLICIT XMLPARSE OPTION special register	"SET CURRENT IMPLICIT XMLPARSE OPTION" on page 1359

Table 69. SQL Session Statements (continued)

SQL Statement	Description	Page
SET ENCRYPTION PASSWORD	Assigns a value to the default encryption password and default encryption password hint	"SET ENCRYPTION PASSWORD" on page 1366
SET PATH	Assigns a value to the CURRENT PATH special register	"SET PATH" on page 1387
SET SCHEMA	Assigns a value to the CURRENT SCHEMA special register	"SET SCHEMA" on page 1393
SET SESSION AUTHORIZATION	Changes the user of the job and the USER special register	"SET SESSION AUTHORIZATION" on page 1396

Table 70. SQL Embedded Host Language Statements

SQL Statement	Description	Page
BEGIN DECLARE SECTION	Marks the beginning of an SQL declare section	"BEGIN DECLARE SECTION" on page 801
CALL	Calls a procedure	"CALL" on page 803
DECLARE PROCEDURE	Defines an external procedure	"DECLARE PROCEDURE" on page 1106
DECLARE STATEMENT	Declares the names used to identify prepared SQL statements	"DECLARE STATEMENT" on page 1116
DECLARE VARIABLE	Declares a subtype or normalized other than the default for a host variable	"DECLARE VARIABLE" on page 1118
END DECLARE SECTION	Marks the end of an SQL declare section	"END DECLARE SECTION" on page 1165
GET DIAGNOSTICS	Obtains information about the previously executed SQL statement	"GET DIAGNOSTICS" on page 1194
INCLUDE	Inserts declarations into a source program	"INCLUDE" on page 1250
SET OPTION	Establishes the options for processing SQL statements	"SET OPTION" on page 1368
SET RESULT SETS	Identifies the result sets in a procedure	"SET RESULT SETS" on page 1390
SIGNAL	Signals an error or warning condition	"SIGNAL" on page 1407
WHENEVER	Defines actions to be taken on the basis of SQL return codes	"WHENEVER" on page 1425

Table 71. SQL Control Statements

SQL Statement	Description	Page
assignment-statement	Assigns a value to an output parameter or to a local variable	"assignment-statement" on page 1438
CALL	Calls a procedure	"CALL statement" on page 1441
CASE	Selects an execution path based on multiple conditions	"CASE statement" on page 1443

Table 71. SQL Control Statements (continued)

SQL Statement	Description	Page
compound-statement	Groups other statements together in an SQL routine	"compound-statement" on page 1445
FOR	Executes a statement for each row of a table	"FOR statement" on page 1455
GET DIAGNOSTICS	Obtains information about the previously executed SQL statement	"GET DIAGNOSTICS statement" on page 1457
GOTO	Branches to a user-defined label within an SQL routine or trigger	"GOTO statement" on page 1465
IF	Provides conditional execution based on the truth value of a condition	"IF statement" on page 1467
ITERATE	Causes the flow of control to return to the beginning of a labeled loop	"ITERATE statement" on page 1469
LEAVE	Continues execution by leaving a block or loop	"LEAVE statement" on page 1471
LOOP	Repeats the execution of a statement	"LOOP statement" on page 1473
REPEAT	Repeats the execution of a statement	"REPEAT statement" on page 1477
RESIGNAL	Resignals an error or warning condition	"RESIGNAL statement" on page 1479
RETURN	Returns from a routine	"RETURN statement" on page 1483
SIGNAL	Signals an error or warning condition	"SIGNAL statement" on page 1486
WHILE	Repeats the execution of a statement while a specified condition is true	"WHILE statement" on page 1490

How SQL statements are invoked

The SQL statements described in this chapter are classified as *executable* or *nonexecutable*. The *Invocation* section in the description of each statement indicates whether the statement is executable.

An *executable statement* can be invoked in any of the following ways:

- Embedded in an application program
- Dynamically prepared and executed
- Issued interactively

Note: Statements embedded in REXX or processed using RUNSQLSTM are prepared and executed dynamically.

Depending on the statement, some or all of these methods can be used. The *Invocation* section in the description of each statement tells you which methods can be used.

A *nonexecutable statement* can only be embedded in an application program.

Embedding a statement in an application program

SQL statements can be included in a source program that will be submitted to the precompiler by using the CRTSQLCBL, CRTSQLCBLI, CRTSQLCI, CRTSQLCPPI, CRTSQLPLI, CRTSQLRPG, or CRTSQLRPGI commands. Such statements are said to be *embedded* in the program.

An embedded statement can be placed anywhere in the program where a host language statement is allowed. Each embedded statement must be preceded by a keyword (or keywords) to indicate that the statement is an SQL statement:

- In C, COBOL, PL/I, and RPG, each embedded statement must be preceded by the keywords EXEC and SQL.
- In Java, each embedded statement must be preceded by the keywords #sql.
- In REXX, each embedded statement must be preceded by the keyword EXECSQL.

Executable statements

An executable statement embedded in an application program is executed every time a statement of the host language would be executed if specified in the same place. This means that a statement within a loop is executed every time the loop is executed, and a statement within a conditional construct is executed only when the condition is satisfied.

An embedded statement can contain references to variables. A variable referenced in this way can be used in two ways:

- As input (the current value of the variable is used in the execution of the statement)
- As output (the variable is assigned a new value as a result of executing the statement)

In particular, all references to variables in expressions and predicates are effectively replaced by current values of the variables; that is, the variables are used as input. The treatment of other references is described individually for each statement.

Follow all executable statements with a test of the SQL return state or the SQL return code. Alternatively, the WHENEVER statement (which is itself nonexecutable) can be used to change the flow of control immediately after the execution of an embedded statement.

Objects referenced in SQL statements need not exist when the statements are prepared.

Nonexecutable statements

An embedded nonexecutable statement is processed only by the precompiler. The precompiler reports any errors encountered in the statement. The statement is *never* executed, and acts as a no-operation if placed among executable statements of the application program. Therefore, do not follow such statements by a test of an SQL return code.

Dynamic preparation and execution

An application program can dynamically build an SQL statement in the form of a character string placed in a variable. In general, the statement is built from some data available to the program (for example, input from a workstation).

The statement can be prepared for execution using the (embedded) statement PREPARE and executed by the (embedded) statement EXECUTE. Alternatively, the (embedded) statement EXECUTE IMMEDIATE can be used to prepare and execute a statement in one step. In Java, the statement can be prepared for execution by means of the Statement, PreparedStatement, and CallableStatement classes, and executed by means of their respective execute() methods.

A statement that is dynamically prepared must not contain references to host variables. Instead, the statement can contain parameter markers. See "PREPARE" on page 1293 for rules concerning the parameter markers. When the prepared statement is executed, the parameter markers are effectively replaced by the current values of the variables specified in the EXECUTE statement. See "EXECUTE" on page 1166 for rules concerning this replacement. After a statement is prepared, it can be executed several times with different values of variables. Parameter markers are not allowed in EXECUTE IMMEDIATE.

In C, COBOL, PL/I, REXX, and RPG, the successful or unsuccessful execution of the statement is indicated by the values returned in the stand-alone SQLCODE or SQLSTATE after the EXECUTE (or EXECUTE IMMEDIATE) statement. The SQL return code should be checked as described above for embedded statements. See the topic "SQL diagnostic information" on page 708 for more information. In Java, the successful or unsuccessful execution of the statement is handled by Java Exceptions.

Static invocation of a select-statement

A *select-statement* can be included as a part of the (nonexecutable) statement DECLARE CURSOR.

Such a statement is executed every time the cursor is opened by means of the (embedded) statement OPEN. After the cursor is open, the result table can be retrieved one row at a time by successive executions of the FETCH statement or multiple rows at a time by using the multiple-row FETCH statement.

Used in this way, the *select-statement* can contain references to variables. These references are effectively replaced by the values that the variables have at the moment of executing OPEN.

Dynamic invocation of a select-statement

An application program can dynamically build a *select-statement* in the form of a character string placed in a variable.

In general, the statement is built from some data available to the program (for example, a query obtained from a workstation). The statement is then executed every time the cursor is opened by means of the (embedded) statement OPEN. After the cursor is open, the result table can be retrieved one row at a time by successive executions of the FETCH statement or multiple rows at a time by using the multiple-row FETCH statement.

Used in this way, the *select-statement* must not contain references to variables. It can instead contain parameter markers. See "PREPARE" on page 1293 for rules concerning the parameter markers. The parameter markers are effectively replaced by the values of the variables specified in the OPEN statement. See "OPEN" on page 1287 for rules concerning this replacement.

Interactive invocation

A capability for entering SQL statements from a workstation is part of the architecture of the database manager. A statement entered in this way is said to be issued interactively.

A statement issued interactively must be an executable statement that does not contain parameter markers or references to variables, because these make sense only in the context of an application program.

The DB2 for i database provides the Start Structured Query Language (STRSQL) command, the Start Query Manager (STRQM) command, and the Run SQL Script support of System i Navigator for this facility. Other products are also available.

SQL diagnostic information

The database manager uses a diagnostics area to store status information and diagnostic information about the execution of an executable SQL statement. When an SQL statement other than GET DIAGNOSTICS or *compound-statement* is processed, the current diagnostics area is cleared, before processing the SQL statement. As each SQL statement is processed, information about the execution of that SQL statement is recorded in the current diagnostics area as one or more completion conditions or exception conditions.

A completion condition indicates the SQL statement completed successfully, completed with a warning condition, or completed with a not found condition. An exception condition indicates that the statement was not successful. The GET DIAGNOSTICS statement can be used in most languages to return conditions and other information about the previously executed SQL statement from the diagnostics area. For more information, see "GET DIAGNOSTICS" on page 1194. Additionally, the condition information is provided through language specific mechanisms:

- For SQL procedures, SQL functions, and SQL triggers, see "SQL-procedure-statement" on page 1435.
- For host languages, see "Detecting and processing error and warning conditions in host language applications."

Detecting and processing error and warning conditions in host language applications

Each host language provides a mechanism for handling diagnostic information:

- In C, COBOL, and PL/I, an application program containing executable SQL statements must provide at least one of the following:
  - A structure named SQLCA.
  - A stand-alone CHAR(5) (CHAR(6) in C) variable named SQLSTATE.
  - A stand-alone integer variable named SQLCODE.

  A stand-alone SQLSTATE or SQLCODE must not be declared in a host structure. Both a stand-alone SQLSTATE and SQLCODE may be provided.

  An SQLCA can be obtained by using the INCLUDE SQLCA statement. If an SQLCA is provided, neither a stand-alone SQLSTATE or SQLCODE can be provided. The SQLCA includes a character-string variable named SQLSTATE and an integer variable named SQLCODE.

A stand-alone SQLSTATE should be used to conform with the SQL 2003 Core standard.

- In Java, for error conditions, the getSQLState method can be used to get the SQLSTATE and the getErrorCode method can be used to get the SQLCODE.
- In REXX and RPG, an SQLCA is provided automatically.

SQLSTATE

The database manager sets SQLSTATE after each SQL statement (other than GET DIAGNOSTICS or a compound statement) is executed. Thus, application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE.

SQLSTATE provides application programs with common codes for common error conditions. Furthermore, SQLSTATE is designed so that application programs can test for specific errors or classes of errors. The scheme is the same for all database managers and is based on the ISO/ANSI SQL 2003 Core standard. A complete list of SQLSTATE classes and SQLSTATEs associated with each SQLCODE is supplied in the SQL Messages and Codes topic collection.

SQLCODE

The database manager sets SQLCODE after each SQL statement (other than GET DIAGNOSTICS or a compound statement) is executed. SQLCODE is set as follows:

- If SQLCODE = 0 and SQLWARN0 is blank, execution was successful.
- If SQLCODE = 100, no data was found. For example, a FETCH statement returned no data, because the cursor was positioned after the last row of the result table.
- If SQLCODE > 0 and not = 100, execution was successful with a warning.
- If SQLCODE = 0 and SQLWARN0 = 'W', execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

A complete listing of DB2 for i SQLCODEs and their corresponding SQLSTATEs is provided in the SQL Messages and Codes topic collection.

SQL comments

In most host languages, static SQL statements can include host language or SQL comments. In Java and REXX, static SQL statements cannot include host language or SQL comments.

Dynamic SQL statements can include SQL comments.

There are two types of SQL comments:

simple comments
Simple comments are introduced by two consecutive hyphens.

bracketed comments
Bracketed comments are introduced by /* and end with */.

These rules apply to the use of simple comments:

- The two hyphens must be on the same line and must not be separated by a space.

- Simple comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Simple comments cannot be continued to the next line.
- In COBOL, the hyphens must be preceded by a space.

These rules apply to the use of bracketed comments:

- The /* must be on the same line and not separated by a space.
- The */ must be on the same line and not separated by a space.
- Bracketed comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Bracketed comments can be continued to the next line.
- Bracketed comments can be nested within other bracketed comments.

A comment embedded in an SQL statement that precedes a name (such as a table name) may cause object names in the text saved for a view, trigger, variable, or MQT to not be maintained correctly. Similarly, names in rows of a dependency view (such as SYSTRIGDEP) may not be correctly qualified.

Example 1

This example shows how to include simple comments in a statement:

```
CREATE VIEW PRJ_MAXPER         -- PROJECTS WITH MOST SUPPORT PERSONNEL
   AS SELECT PROJNO, PROJNAME  -- NUMBER AND NAME OF PROJECT
      FROM PROJECT
      WHERE DEPTNO = 'E21'     -- SYSTEMS SUPPORT DEPT CODE
      AND PRSTAFF > 1
```

Example 2

This example shows how to include bracketed comments in a statement:

```
CREATE VIEW PRJ_MAXPER         /* PROJECTS WITH MOST SUPPORT
                                     PERSONNEL               */
   AS SELECT PROJNO, PROJNAME  /* NUMBER AND NAME OF PROJECT   */
      FROM PROJECT
      WHERE DEPTNO = 'E21'     /* SYSTEMS SUPPORT DEPT CODE    */
      AND PRSTAFF > 1
```

ALLOCATE CURSOR

The ALLOCATE CURSOR statement defines a cursor and associates it with a result set locator variable.

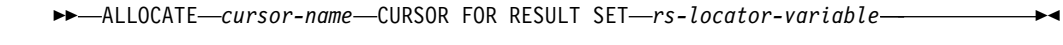
Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared. It cannot be issued interactively. It must not be specified in REXX.

Authorization

None required.

Syntax



Description

cursor-name
Names the cursor. The name must not identify a cursor that has already been declared in the source program.

CURSOR FOR RESULT SET ***rs-locator-variable***
Specifies a result set locator variable that has been declared in the application program according to the rules for declaring result set locator variables.

The result set locator variable must contain a valid result set locator value as returned by the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE SQL statement. The value of the result set locator variable is used at the time the cursor is allocated. Subsequent changes to the value of the result set locator have no effect on the allocated cursor. The result set locator value must not be the same as a value used for another cursor allocated in the source program.

Rules

- The following rules apply when you use an allocated cursor:
  - You cannot open an allocated cursor with the OPEN statement.
  - You can close an allocated cursor with the CLOSE statement. Closing an allocated cursor closes the associated cursor defined in the stored procedure.
  - You can allocate only one cursor to each result set.
- Allocated cursors follow the same rules as declared cursors in a program with the CLOSQLCSR option. The CLOSQLCSR option can be specified on the CRTSQLxxx command or using the SET OPTION statement.
- A commit operation closes allocated cursors that are not defined WITH HOLD by the procedure and are not running with a commit level other than *NONE.
- Closing an allocated cursor closes the associated cursor in the procedure.

Example

This SQL procedure example defines and associates cursor C1 with the result set locator variable LOC1 and the related result set returned by the SQL procedure:

```
ALLOCATE C1 CURSOR FOR RESULT SET LOC1;
```

ALLOCATE DESCRIPTOR

The ALLOCATE DESCRIPTOR statement allocates an SQL descriptor.

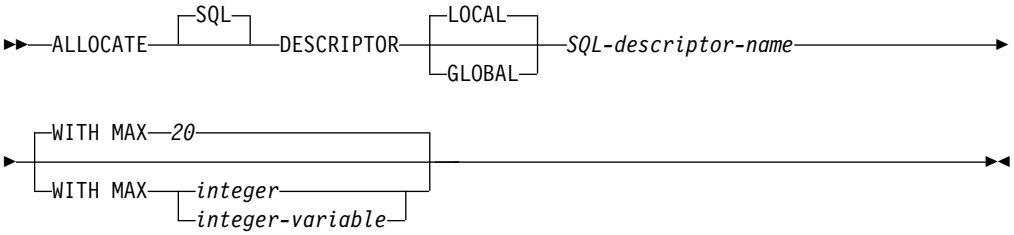
Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It cannot be issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

Authorization

None required.

Syntax



Description

LOCAL
Defines the scope of the name of the descriptor to be local to the program invocation. The descriptor will not be known outside this scope. For example, a program called from another separately compiled program cannot use a descriptor that was allocated by the calling program. The scope of the descriptor is also limited to the thread in which the program that contains the descriptor is running. For example, if the same program is running in two separate threads in the same job, the second thread cannot use a descriptor that was allocated by the first thread.

GLOBAL
Defines the scope of the name of the descriptor to be global to the SQL session. The descriptor will be known to any program that executes using the same database connection.

SQL-descriptor-name
Names the descriptor to allocate. The name must not be the same as a descriptor that already exists with the specified scope.

WITH MAX
The descriptor is allocated to support the specified maximum number of items. If this clause is not specified, the descriptor is allocated with a maximum of 20 items.

integer
Specifies the number of items to allocate. The value of *integer* must be greater than zero and not greater than 8000.

integer-variable
Specifies an integer variable (or decimal or numeric variable with zero

scale) that contains the number of items to allocate. It cannot be a global variable. The value of *integer-variable* must be greater than zero and not greater than 8000.

Notes

Descriptor persistence: Local descriptors are implicitly deallocated based on the CLOSQLCSR option:

- For ILE programs, if CLOSQLCSR(*ENDACTGRP) is specified (the default), local descriptors are implicitly deallocated when the activation group ends. If CLOSQLCSR(*ENDMOD) is specified, local descriptors are implicitly deallocated on exit from the module.
- For OPM programs, if CLOSQLCSR(*ENDPGM) is specified (the default), local descriptors are implicitly deallocated when the program ends. If CLOSQLCSR(*ENDSQL) is specified, local descriptors are implicitly deallocated when the first SQL program on the call stack ends. If CLOSQLCSR(*ENDJOB) is specified, local descriptors are implicitly deallocated when the job ends.

Global descriptors are implicitly deallocated when the activation group ends.

Both local and global descriptors can be explicitly deallocated using the DEALLOCATE DESCRIPTOR statement.

Examples

Allocate a descriptor called 'NEWDA' large enough to hold 20 items.

```
EXEC SQL ALLOCATE DESCRIPTOR 'NEWDA'
  WITH MAX 20
```

ALTER FUNCTION (External Scalar)

The ALTER FUNCTION (External Scalar) statement alters an external scalar function at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

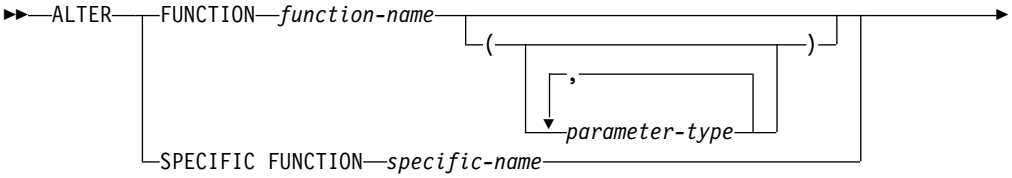
The privileges held by the authorization ID of the statement must include at least one of the following:

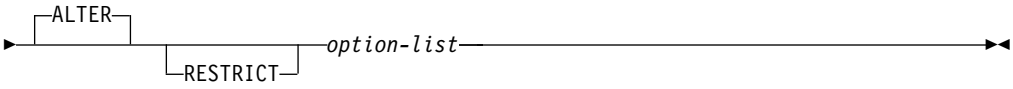
- For the function identified in the statement:
  - The ALTER privilege for the function, and
  - The system authority *EXECUTE on the library containing the function.
- Administrative authority

If a different external program is specified, the privileges held by the authorization ID of the statement must also include the same privileges required to create a new external scalar function. For more information, see "CREATE FUNCTION (External Scalar)" on page 856.

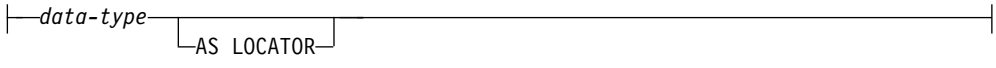
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure, Corresponding System Authorities When Checking Privileges to a Table or View, and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax



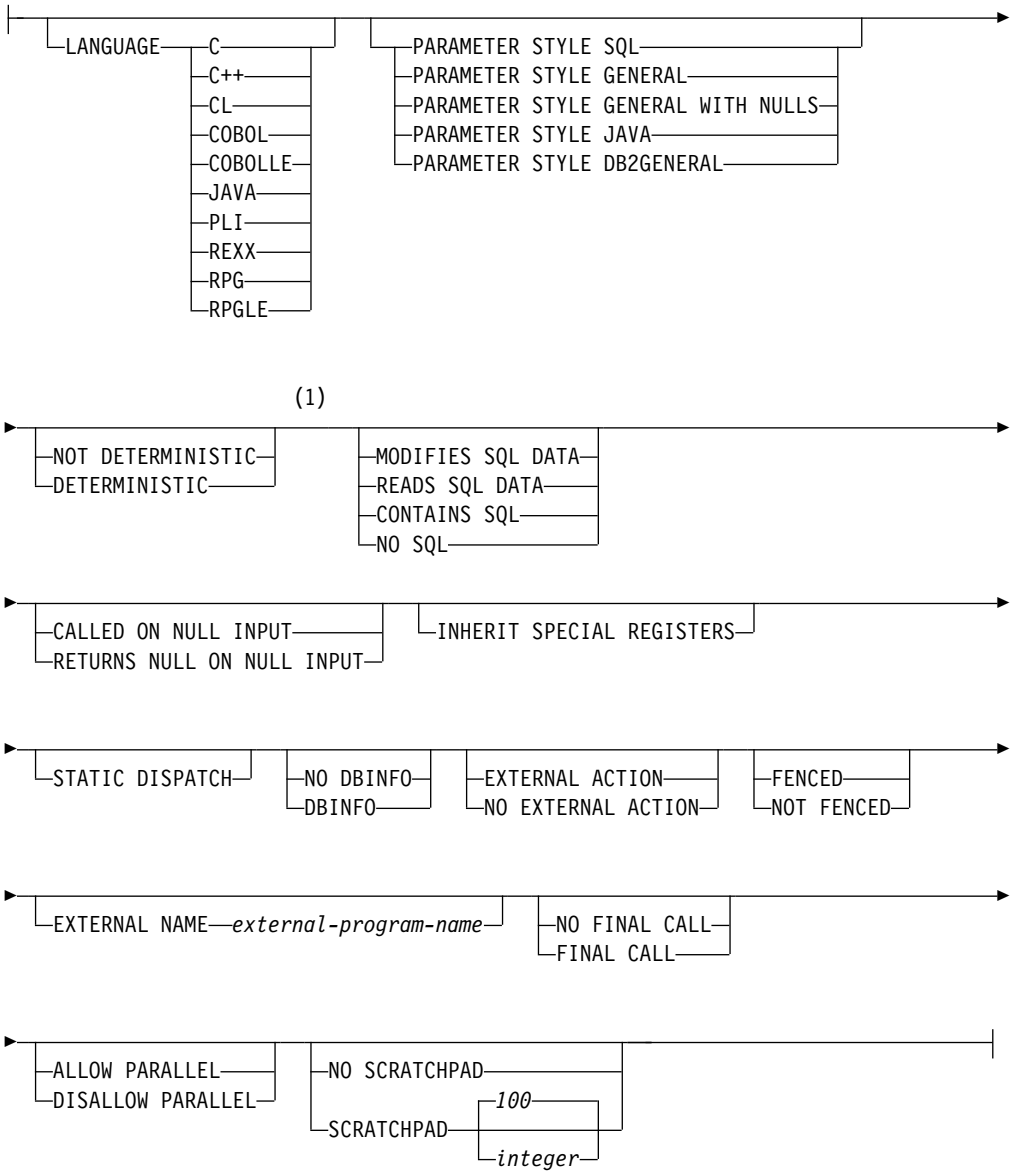


parameter-type:



data-type:

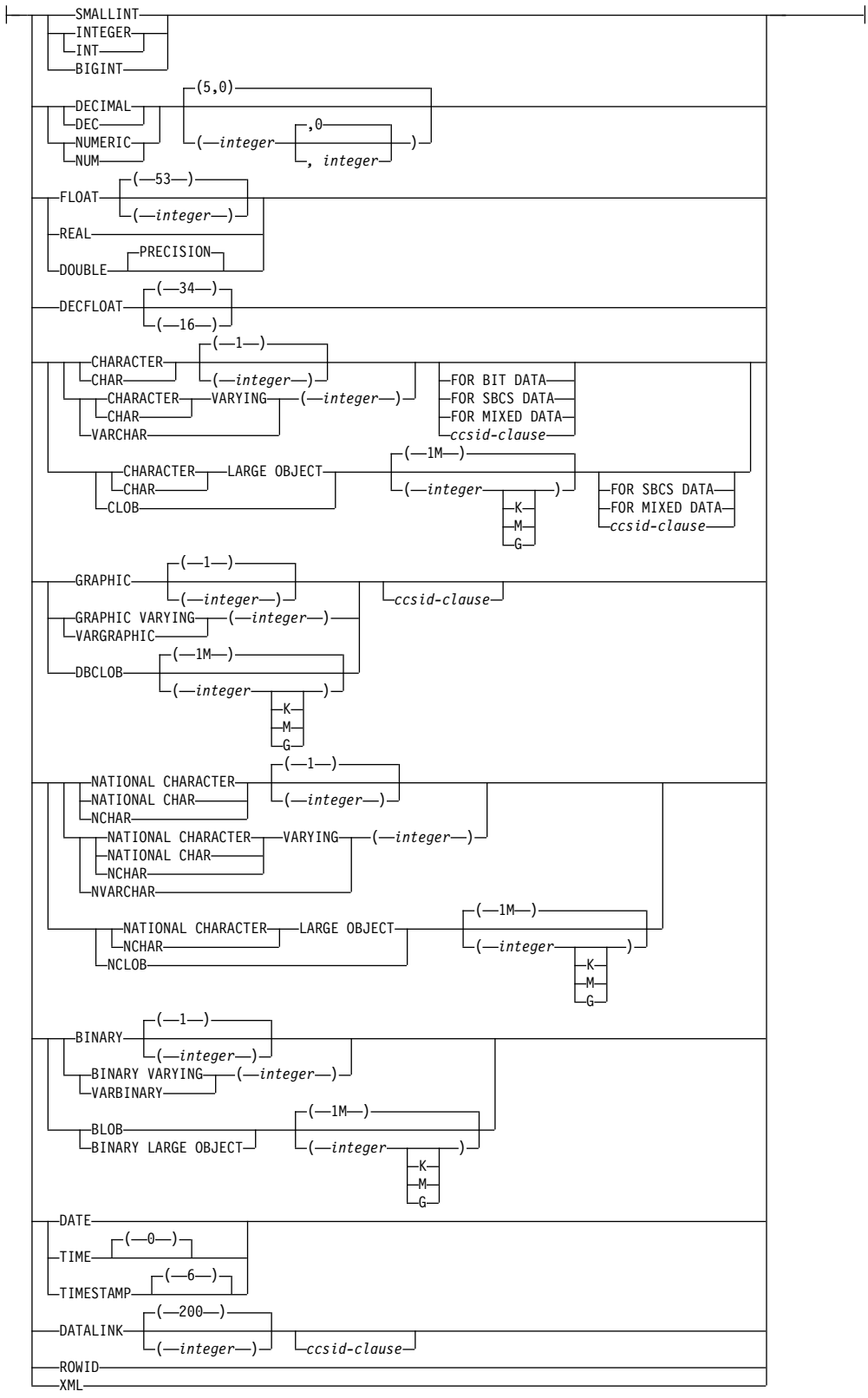
option-list:



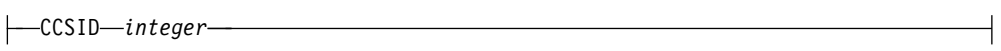
Notes:

1 The clauses in the *option-list* can be specified in any order.

built-in-type:



ccsid-clause:



Description

FUNCTION or SPECIFIC FUNCTION

Identifies the function to alter. *function-name* must identify an external scalar function that exists at the current server. It cannot identify a built-in function, a sourced function, or an SQL function. An external table function cannot be altered to be an external scalar function.

The specified function is altered. The owner of the function is preserved. If the external program or service program exists at the time the function is altered, all privileges on the function are preserved.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type,...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type,...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is being altered. Synonyms for data types are considered a match.

If *function-name()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type,...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parenthesis indicates that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its precision value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

ALTER *option-list*
Indicates that one or more of the options of the function are to be altered. If an option is not specified, the value from the existing function definition is used. See "CREATE FUNCTION (External Scalar)" on page 856 for a description of each option.

RESTRICT
Indicates that the function will not be altered if it is referenced by any view, function, procedure, or materialized query table.

Notes

General considerations for defining or replacing functions: See CREATE FUNCTION (External Scalar) for general information about defining a function. ALTER FUNCTION (External Scalar) allows individual attributes to be altered while preserving the privileges on the function.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords SIMPLE CALL can be used as a synonym for GENERAL.
- The keyword DB2GENRL may be used as a synonym for DB2GENERAL.
- The value DB2SQL may be used as a synonym for SQL.
- The keywords PARAMETER STYLE in the PARAMETER STYLE clause are optional.
- The keywords IS DETERMINISTIC may be used as a synonym for DETERMINISTIC.

Example

Modify the definition for function MYFUNC to change the name of the external program that will be invoked when the function is invoked. The name of the external program is PROG10B.

```
ALTER FUNCTION MYFUNC
  EXTERNAL NAME PROG10B
```

ALTER FUNCTION (External Table)

The ALTER FUNCTION (External Table) statement alters an external table function at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

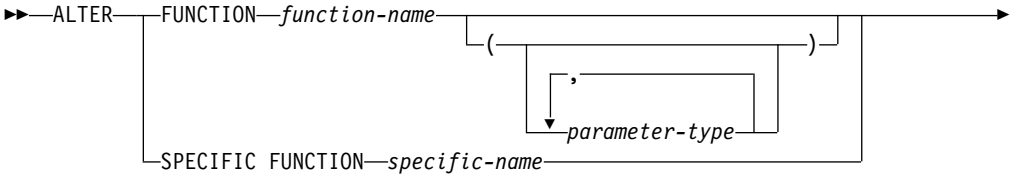
The privileges held by the authorization ID of the statement must include at least one of the following:

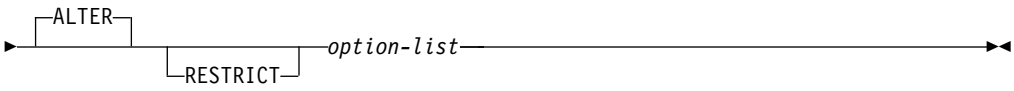
- For the function identified in the statement:
  - The ALTER privilege for the function, and
  - The system authority *EXECUTE on the library containing the function.
- Administrative authority

If a different external program is specified, the privileges held by the authorization ID of the statement must also include the same privileges required to create a new external table function. For more information, see "CREATE FUNCTION (External Table)" on page 876.

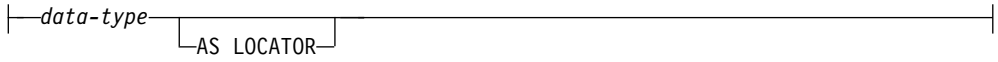
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure, Corresponding System Authorities When Checking Privileges to a Table or View, and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax



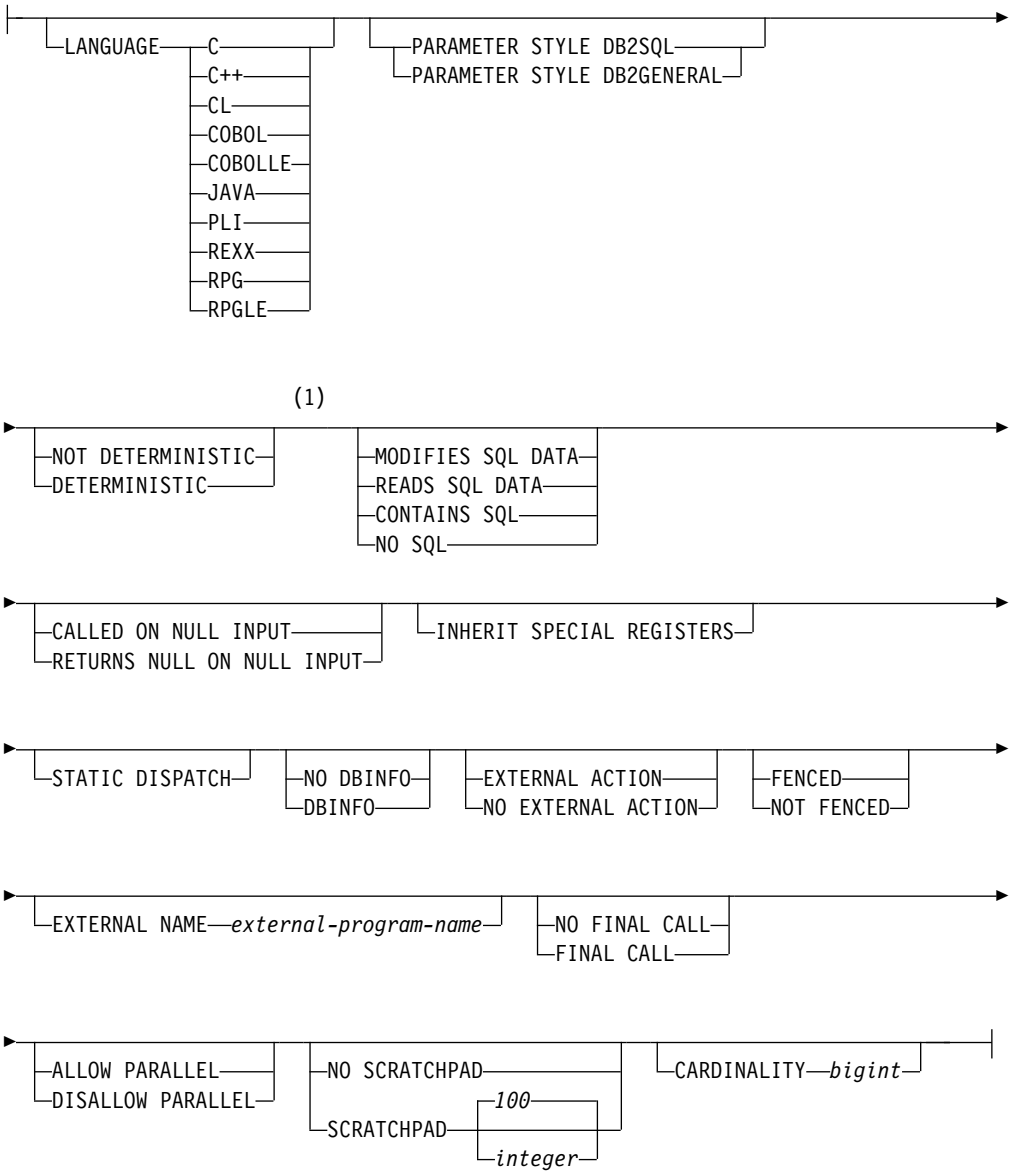


parameter-type:



data-type:

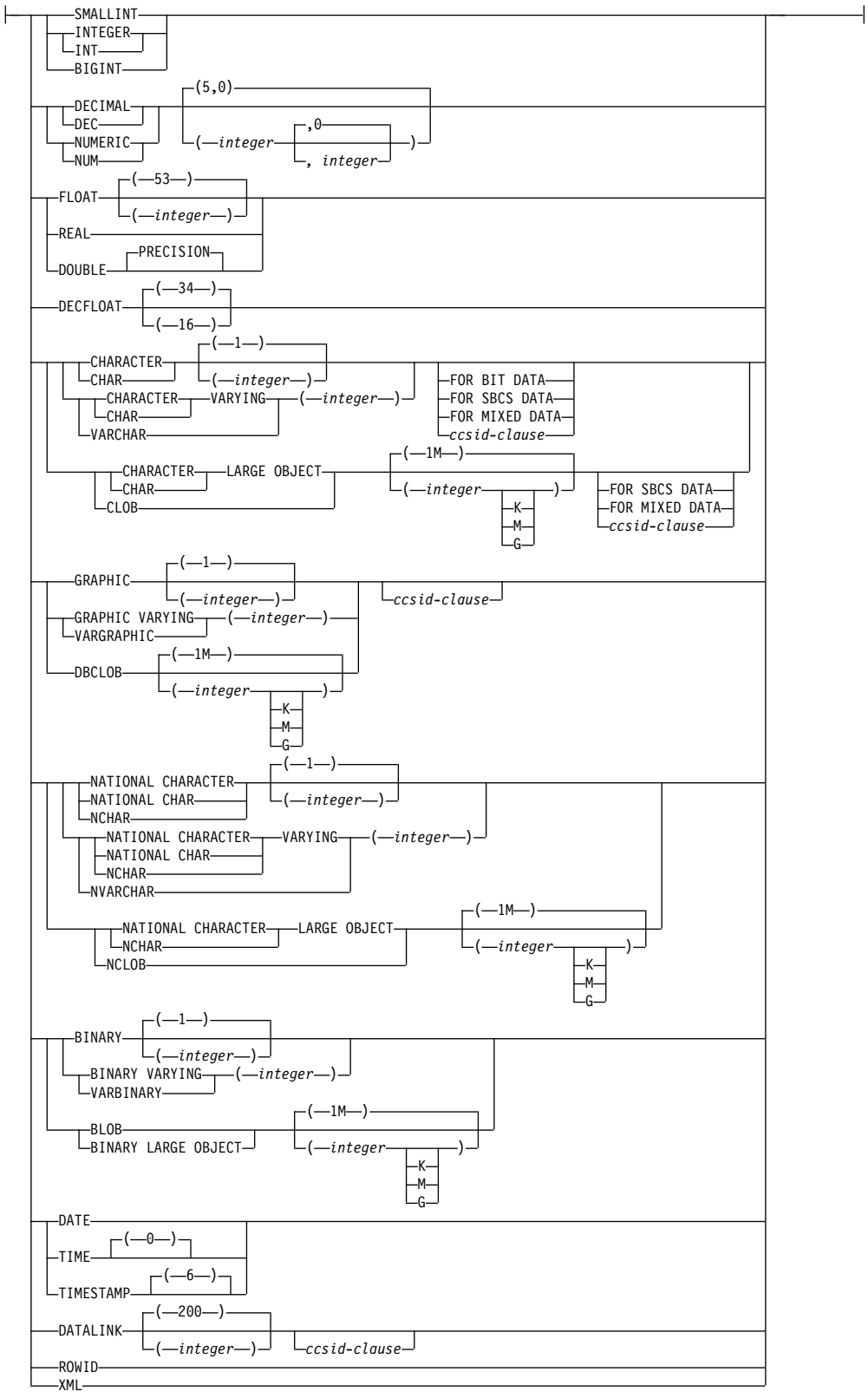
option-list:



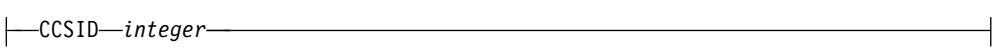
Notes:

1 The clauses in the *option-list* can be specified in any order.

built-in-type:



ccsid-clause:



Description

FUNCTION or SPECIFIC FUNCTION

Identifies the function to alter. *function-name* must identify an external table function that exists at the current server. It cannot identify a built-in function, a sourced function, or an SQL function. An external scalar function cannot be altered to be an external table function.

The specified function is altered. The owner of the function is preserved. If the external program or service program exists at the time the function is altered, all privileges on the function are preserved.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type,...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type,...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is being altered. Synonyms for data types are considered a match.

If *function-name()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type,...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parenthesis indicates that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its precision value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

ALTER *option-list*
Indicates that one or more of the options of the function are to be altered. If an option is not specified, the value from the existing function definition is used. See "CREATE FUNCTION (External Table)" on page 876 for a description of each option.

RESTRICT
Indicates that the function will not be altered if it is referenced by any view, function, procedure, or materialized query table.

Notes

General considerations for defining or replacing functions: See CREATE FUNCTION (External Table) for general information about defining a function. ALTER FUNCTION (External Table) allows individual attributes to be altered while preserving the privileges on the function.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keyword DB2GENRL may be used as a synonym for DB2GENERAL.
- The keywords PARAMETER STYLE in the PARAMETER STYLE clause are optional.
- The keywords IS DETERMINISTIC may be used as a synonym for DETERMINISTIC.

Example

Modify the definition for an external table function to set the estimated cardinality to 10,000.

```
ALTER FUNCTION GET_TABLE
  ALTER CARDINALITY 10000
```

ALTER FUNCTION (SQL Scalar)

The ALTER FUNCTION (SQL Scalar) statement alters an SQL scalar function at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

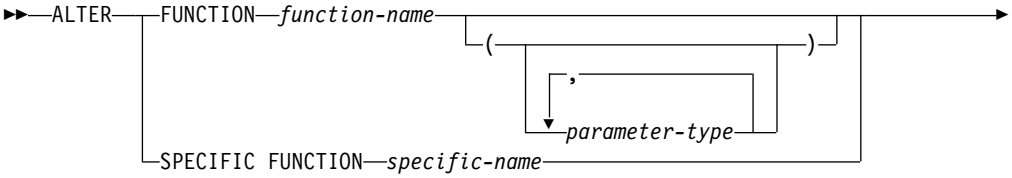
The privileges held by the authorization ID of the statement must include at least one of the following:

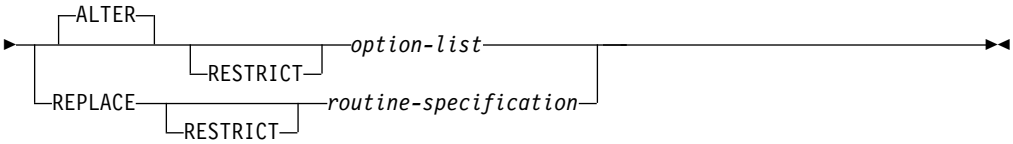
- For the function identified in the statement:
  - The ALTER privilege for the function, and
  - The system authority *EXECUTE on the library containing the function.
- Administrative authority

If a different external program is specified, the privileges held by the authorization ID of the statement must also include the same privileges required to create a new external scalar function. For more information, see "CREATE FUNCTION (SQL Scalar)" on page 905.

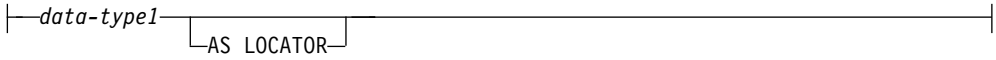
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure, Corresponding System Authorities When Checking Privileges to a Table or View, and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax

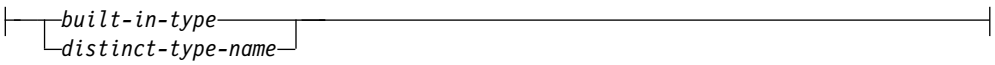




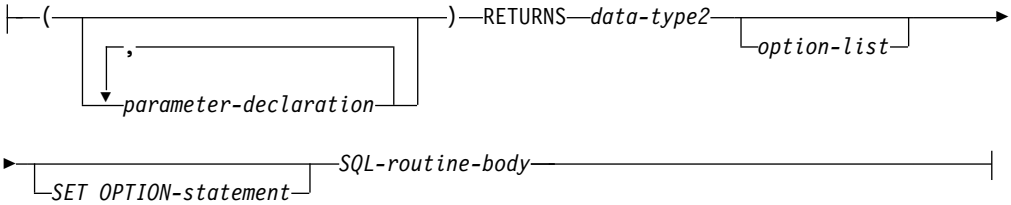
parameter-type:



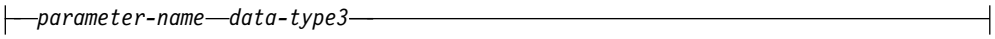
data-type1, data-type2,data-type3:



routine-specification:

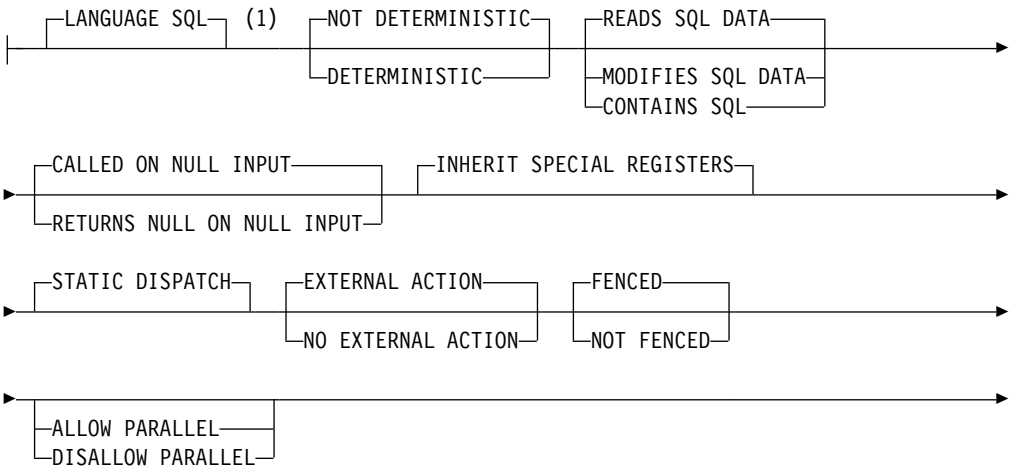


parameter-declaration:



SQL-routine-body:

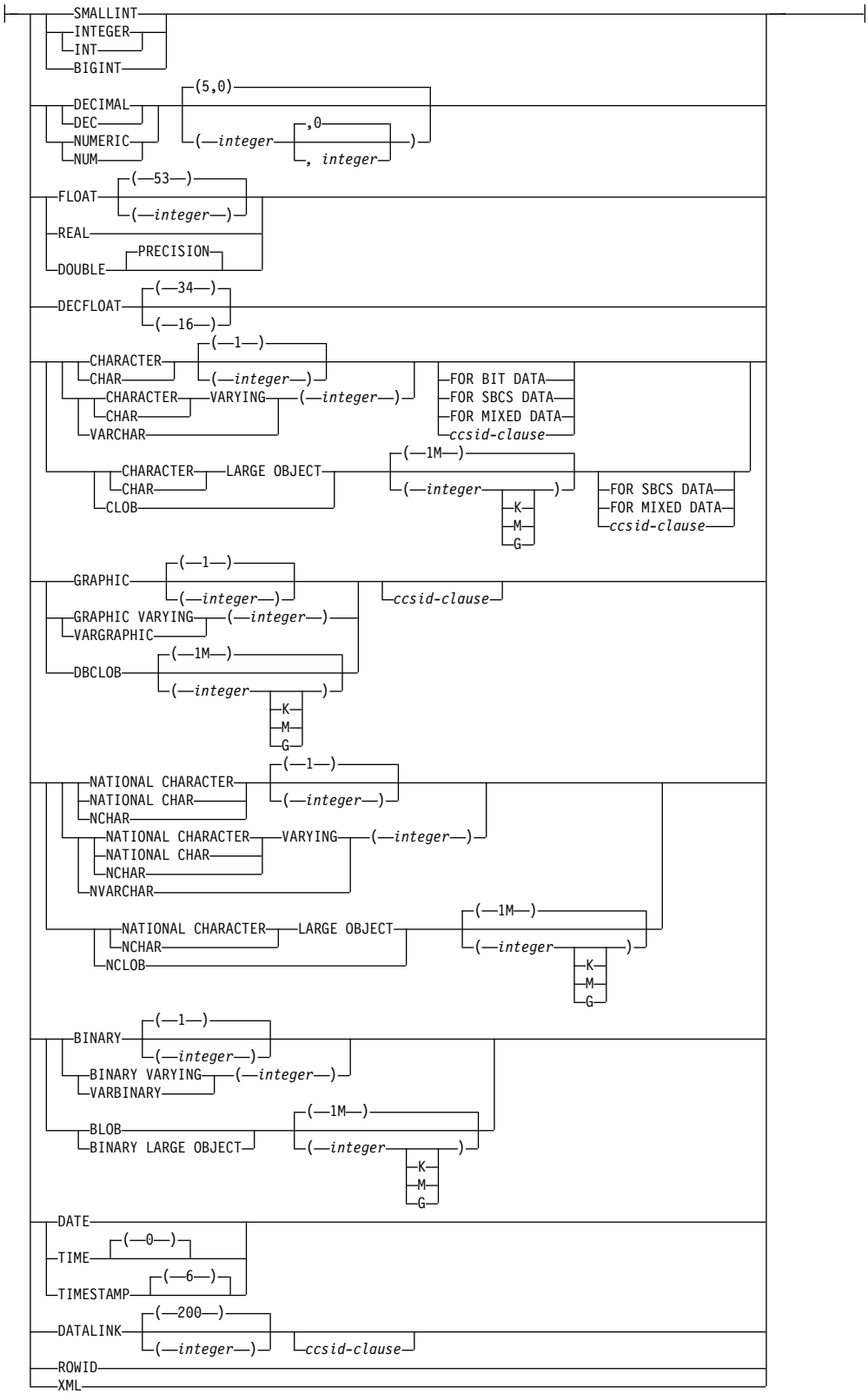
option-list:



Notes:

1 The clauses in the *option-list* can be specified in any order.

built-in-type:



ccsid-clause:

Description

FUNCTION or SPECIFIC FUNCTION
Identifies the function to alter. *function-name* must identify an SQL scalar function that exists at the current server.

The specified function is altered. The owner of the function and all privileges on the function are preserved.

FUNCTION *function-name*
Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type,...)*
Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type,...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is being altered. Synonyms for data types are considered a match.

If *function-name()* is specified, the function identified must have zero parameters.

function-name
Identifies the name of the function.

(parameter-type,...)
Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parenthesis indicates that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its precision value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type

are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

ALTER *option-list*
Indicates that one or more of the options of the function are to be altered. If ALTER FUNCTION ALTER *option-list* is specified and an option is not specified, the value from the existing function definition is used. See "CREATE FUNCTION (SQL Scalar)" on page 905 for a description of each option.

REPLACE *routine-specification*
Indicates that the existing function definition, including options and parameters, is to be replaced by those specified in this statement. The values of all options are replaced when a function is replaced. If an option is not specified, the same default is used as when a new SQL scalar function is created, for more information see "CREATE FUNCTION (SQL Scalar)" on page 905.

If the routine has a comment or label, they are removed from the routine definition.

RESTRICT
Indicates that the function will not be altered or replaced if it is referenced by any function, materialized query table, procedure, trigger, or view.

(*parameter-declaration***,...)**
Specifies the number of parameters of the function, the data type of each parameter, and the name of each parameter.

The maximum number of parameters allowed in an SQL function is 1024.

parameter-name
Names the parameter. The name is used to refer to the parameter within the body of the function. The name cannot be the same as any other *parameter-name* in the parameter list.

data-type3
Specifies the data type of the input parameter. If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

RETURNS
Specifies the output of the function.

data-type2
Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, or LONG VARGRAPHIC) or a distinct type.

If a CCSID is specified and the CCSID of the return data is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified the return data is converted to the CCSID of the job (or associated graphic CCSID of the job for graphic string return values), if the CCSID of the return data is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

option-list
List of options for the function being altered. These options are the same ones that are listed above under ALTER *option-list*. If a specific option is not specified, the same default that is used when a new function is created is used. For more information see "CREATE FUNCTION (SQL Scalar)" on page 905.

SET OPTION-statement
Specifies the options that will be used to create the function. For example, to create a debuggable function, the following statement could be included:

```
SET OPTION DBGVIEW = *SOURCE
```

For more information, see "SET OPTION" on page 1368.

The options CNULRQD, COMPILEOPT, NAMING, and SQLCA are not allowed in the ALTER FUNCTION statement.

SQL-routine-body
Specifies a single SQL statement, including a compound statement. See Chapter 7, "SQL control statements," on page 1427 for more information about defining SQL functions.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK, and SET TRANSACTION statement is not allowed in a function.

The *SQL-routine-body* must contain at least one RETURN statement and a RETURN statement must be executed when the function is called.

ALTER PROCEDURE (SQL), ALTER FUNCTION (SQL Scalar), and ALTER FUNCTION (SQL Table) with a REPLACE keyword are not allowed in an *SQL-routine-body*.

Notes

General considerations for defining or replacing functions: See CREATE FUNCTION (SQL Scalar) for general information about defining a function. ALTER FUNCTION (SQL Scalar) allows individual attributes to be altered while preserving the privileges on the function.

Cascaded effects: If REPLACE is specified without RESTRICT and the function signature or result data type is altered, the results from any function, materialized query table, procedure, trigger, or view that references the function may be unpredictable. Any referenced objects should be recreated.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords IS DETERMINISTIC may be used as a synonym for DETERMINISTIC.

Example

Modify the definition for an SQL scalar function to indicate that the function is deterministic.

```
ALTER FUNCTION MY_UDF1
  DETERMINISTIC
```

ALTER FUNCTION (SQL Table)

The ALTER FUNCTION (SQL Table) statement alters an SQL table function at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

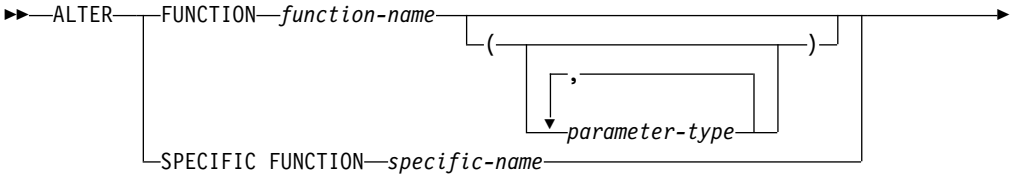
The privileges held by the authorization ID of the statement must include at least one of the following:

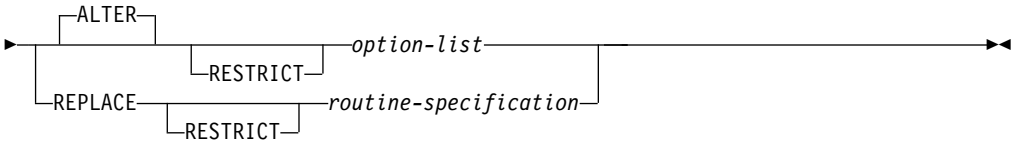
- For the function identified in the statement:
  - The ALTER privilege for the function, and
  - The system authority *EXECUTE on the library containing the function.
- Administrative authority

If a different external program is specified, the privileges held by the authorization ID of the statement must also include the same privileges required to create a new external table function. For more information, see "CREATE FUNCTION (SQL Table)" on page 917.

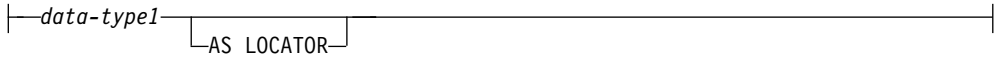
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure, Corresponding System Authorities When Checking Privileges to a Table or View, and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax

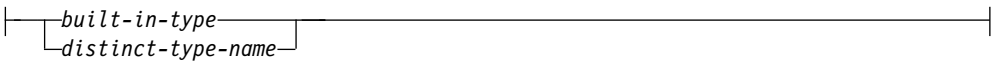




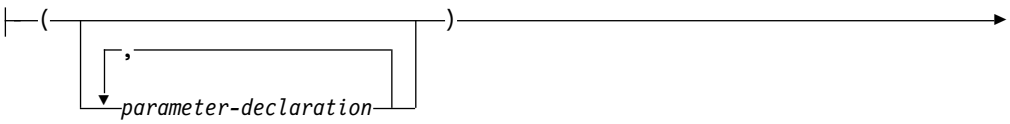
parameter-type:

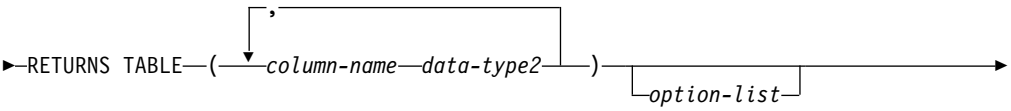


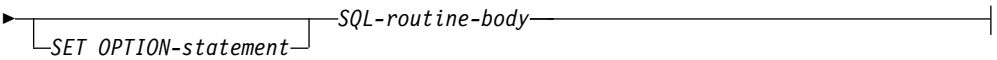
data-type1, data-type2, data-type3:



routine-specification:

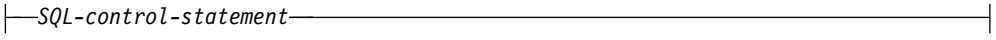




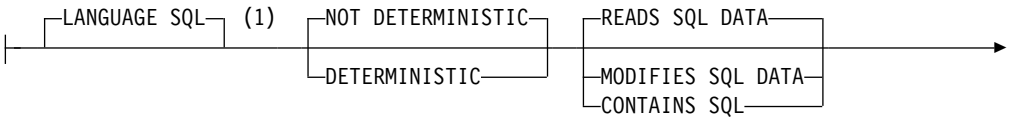


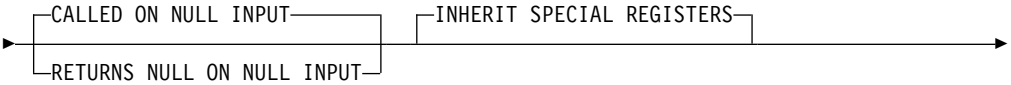
parameter-declaration:

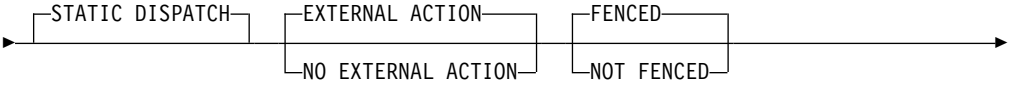
SQL-routine-body:

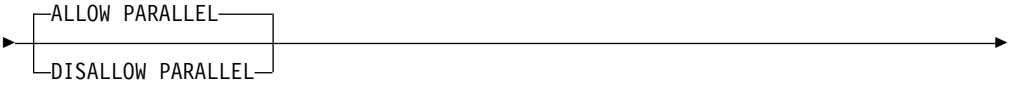


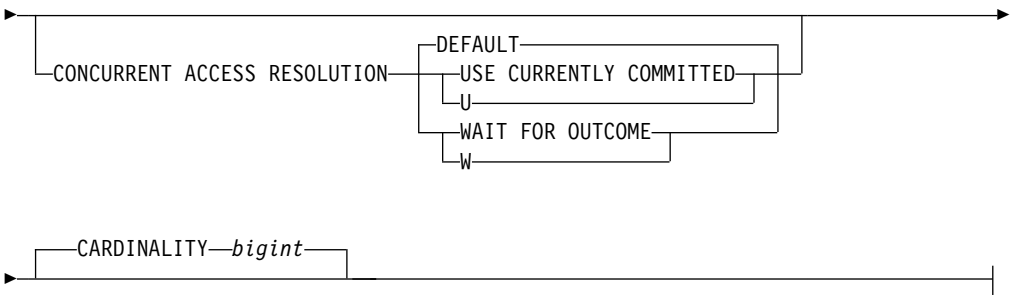
option-list:







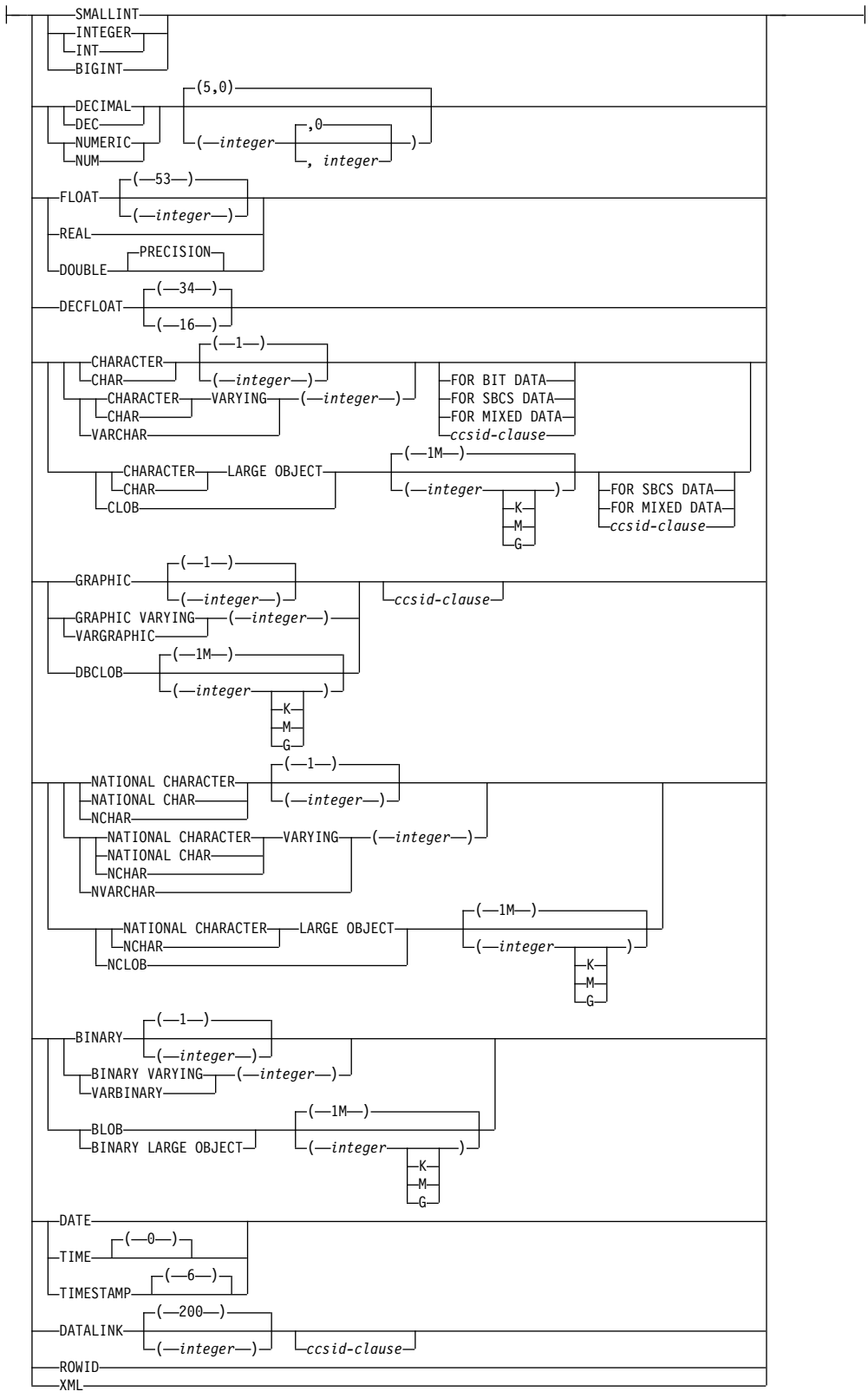




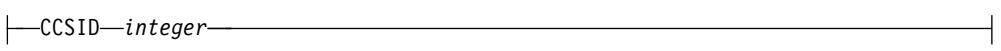
Notes:

1 The clauses in the *option-list* can be specified in any order.

built-in-type:



ccsid-clause:



Description

FUNCTION or SPECIFIC FUNCTION

Identifies the function to alter. *function-name* must identify an SQL table function that exists at the current server.

The specified function is altered. The owner of the function and all privileges on the function are preserved.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type,...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type,...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is being altered. Synonyms for data types are considered a match.

If *function-name()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type,...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parenthesis indicates that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its precision value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either

clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

ALTER *option-list*
Indicates that one or more of the options of the function are to be altered. If ALTER FUNCTION ALTER *option-list* is specified and an option is not specified, the value from the existing function definition is used. See "CREATE FUNCTION (SQL Table)" on page 917 for a description of each option.

REPLACE *routine-specification*
Indicates that the existing function definition, including options and parameters, is to be replaced by those specified in this statement. The values of all options are replaced when a function is replaced. If an option is not specified, the same default is used as when a new SQL table function is created, for more information see "CREATE FUNCTION (SQL Table)" on page 917.

If the routine has a comment or label, they are removed from the routine definition.

RESTRICT
Indicates that the function will not be altered or replaced if it is referenced by any function, materialized query table, procedure, trigger, or view.

(*parameter-declaration***,...)**
Specifies the number of parameters of the function, the data type of each parameter, and the name of each parameter.

The maximum number of parameters allowed in an SQL function is 1024.

parameter-name
Names the parameter. The name is used to refer to the parameter within the body of the function. The name cannot be the same as any other *parameter-name* in the parameter list.

data-type3
Specifies the data type of the input parameter. If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

RETURNS TABLE
Specifies the output table of the function.

Assume the number of parameters is N. There must be no more than 1025 - *N* columns.

column-name
Specifies the name of a column of the output table. Do not specify the same name more than once.

data-type2
Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, or LONG VARGRAPHIC) or a distinct type.

If a CCSID is specified and the CCSID of the return data is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified the return data is converted to the CCSID of the job (or associated graphic CCSID of the job for graphic string return values), if the CCSID of the return data is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

option-list
List of options for the function being altered. These options are the same ones that are listed above under ALTER *option-list*. If a specific option is not specified, the same default that is used when a new function is created is used. For more information see "CREATE FUNCTION (SQL Table)" on page 917.

SET OPTION-statement
Specifies the options that will be used to create the function. For example, to create a debuggable function, the following statement could be included:

```
SET OPTION DBGVIEW = *SOURCE
```

For more information, see "SET OPTION" on page 1368.

The options CNULRQD, COMPILEOPT, NAMING, and SQLCA are not allowed in the ALTER FUNCTION statement.

SQL-routine-body
Specifies a single SQL statement, including a compound statement. See Chapter 7, "SQL control statements," on page 1427 for more information about defining SQL functions.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK, and SET TRANSACTION statement is not allowed in a function.

The *SQL-routine-body* must contain at least one RETURN statement and a RETURN statement must be executed when the function is called.

ALTER PROCEDURE (SQL), ALTER FUNCTION (SQL Scalar), and ALTER FUNCTION (SQL Table) with a REPLACE keyword are not allowed in an *SQL-routine-body*.

Notes

General considerations for defining or replacing functions: See CREATE FUNCTION (SQL Table) for general information about defining a function. ALTER FUNCTION (SQL Table) allows individual attributes to be altered while preserving the privileges on the function.

Cascaded effects: If REPLACE is specified without RESTRICT and the function signature or result data types are altered, the results from any function, materialized query table, procedure, trigger, or view that references the function may be unpredictable. Any referenced objects should be recreated.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords IS DETERMINISTIC may be used as a synonym for DETERMINISTIC.

Example

Modify the definition for an SQL table function to set the estimated cardinality to 10,000.

```
ALTER FUNCTION GET_TABLE
  ALTER CARDINALITY 10000
```

ALTER PROCEDURE (External)

The ALTER PROCEDURE (External) statement alters an external procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

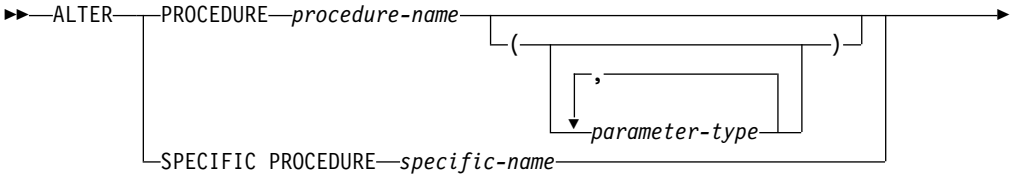
The privileges held by the authorization ID of the statement must include at least one of the following:

- For the procedure identified in the statement:
  - The ALTER privilege for the procedure, and
  - The system authority *EXECUTE on the library containing the procedure.
- Administrative authority

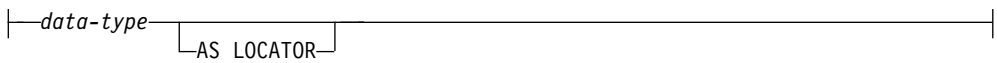
If a different external program is specified, the privileges held by the authorization ID of the statement must also include the same privileges required to create a new external procedure. For more information, see "CREATE PROCEDURE (External)" on page 939.

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure, Corresponding System Authorities When Checking Privileges to a Table or View, and Corresponding System Authorities When Checking Privileges to a Distinct Type.

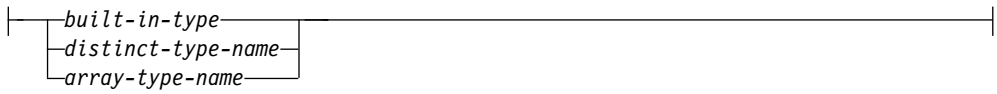
Syntax



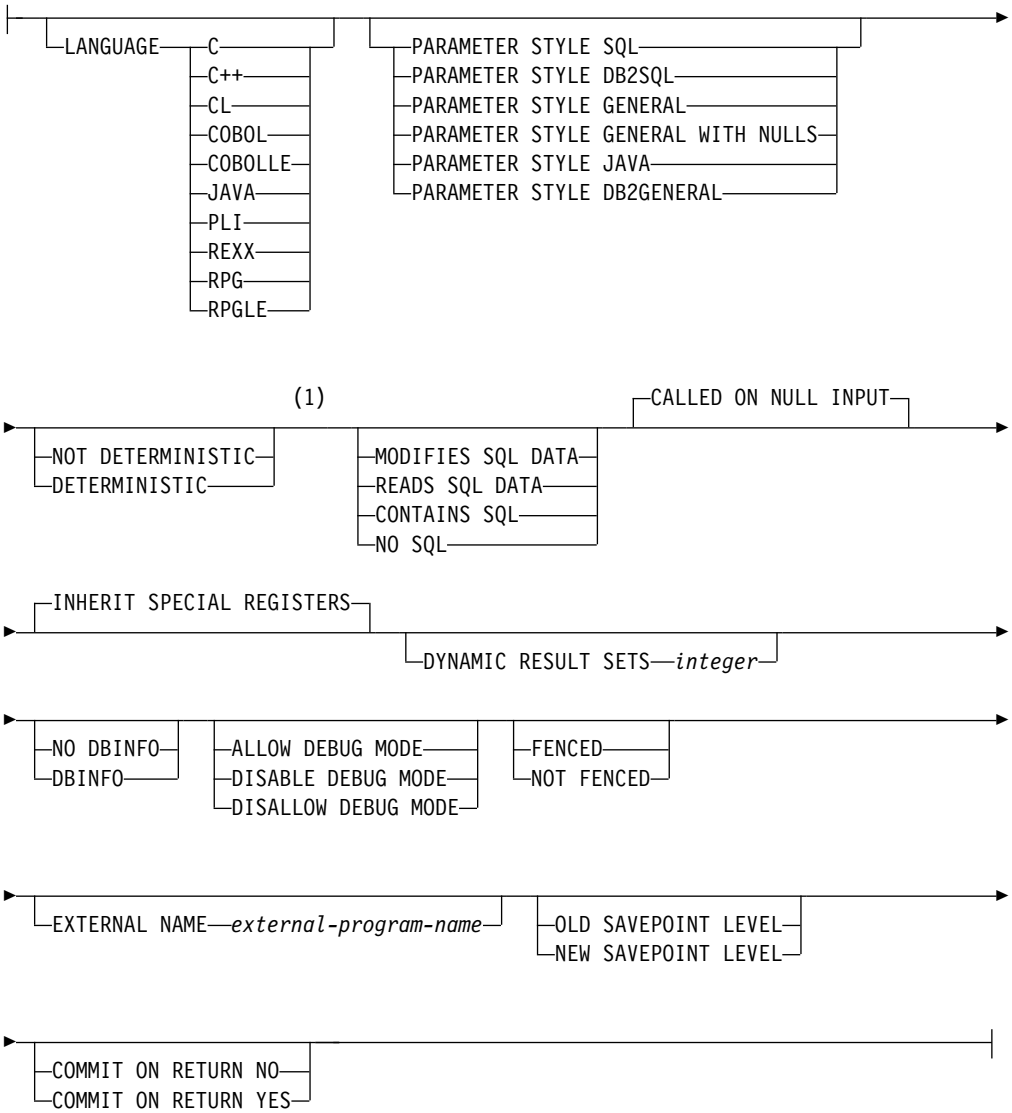
parameter-type:



data-type:



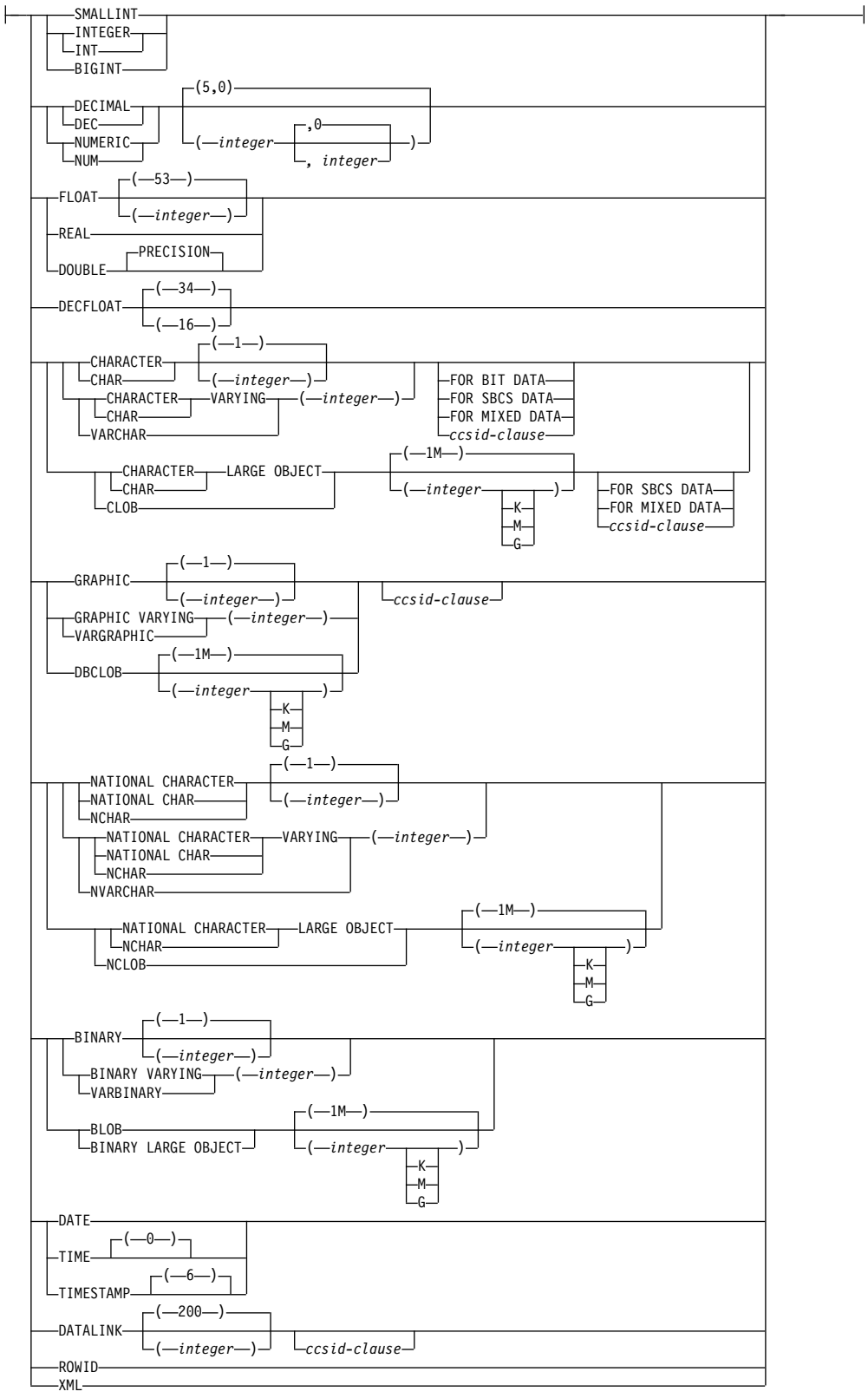
option-list:



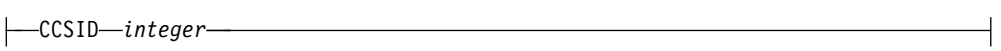
Notes:

1 The clauses in the *option-list* can be specified in any order.

built-in-type:



ccsid-clause:



Description

PROCEDURE or SPECIFIC PROCEDURE

Identifies the procedure to alter. *procedure-name* must identify an external procedure that exists at the current server.

The specified procedure is altered. The owner of the procedure is preserved. If the external program or service program exists at the time the procedure is altered, all privileges on the procedure are preserved.

PROCEDURE *procedure-name*

Identifies the procedure by its name. The *procedure-name* must identify exactly one external procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type,...)*

Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type,...)* must identify an external procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types and the logical concatenation of the data types is used to identify the specific procedure instance which is being altered. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name()* is specified, the procedure identified must have zero parameters.

procedure-name

Identifies the name of the procedure.

(parameter-type,...)

Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parenthesis indicates that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its precision value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

ALTER *option-list*
Indicates that one or more of the options of the procedure are to be altered. If an option is not specified, the value from the existing procedure definition is used. See "CREATE PROCEDURE (External)" on page 939 for a description of each option.

Notes

General considerations for defining or changing a procedure: See CREATE PROCEDURE for general information about defining a procedure. ALTER PROCEDURE (External) allows individual attributes to be altered while preserving the privileges on the procedure.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL can be used as synonyms for CALLED ON NULL INPUT.
- DYNAMIC RESULT SET, RESULT SETS, and RESULT SET may be used as synonyms for DYNAMIC RESULT SETS.

Examples

Modify the definition for procedure MYPROC to change the name of the external program that is invoked when the procedure is called. The name of the external program is PROG10A.

```
ALTER PROCEDURE MYPROC
  EXTERNAL NAME PROG10A
```

ALTER PROCEDURE (SQL)

The ALTER PROCEDURE (SQL) statement alters a procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

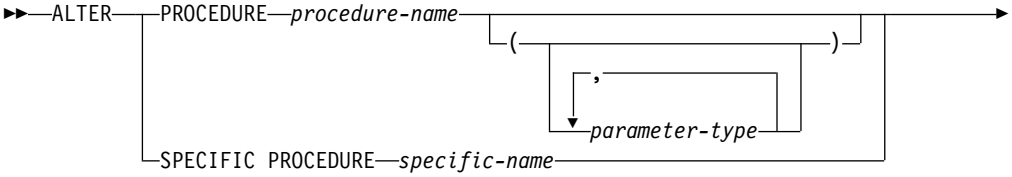
- For the procedure identified in the statement:
  - The ALTER privilege for the procedure, and
  - The system authority *EXECUTE on the library containing the procedure.
- Administrative authority

If a distinct type is referenced in a *parameter-declaration*, the privileges held by the authorization ID of the statement must include at least one of the following:

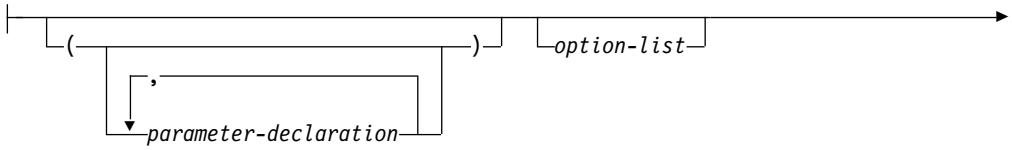
- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

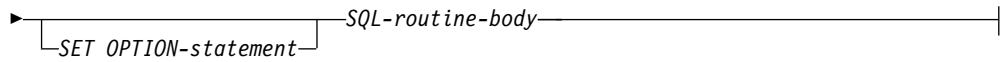
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax

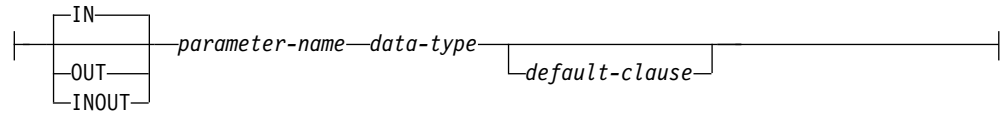


routine-specification:

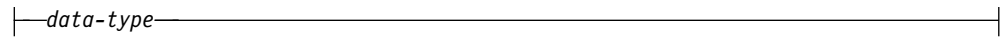




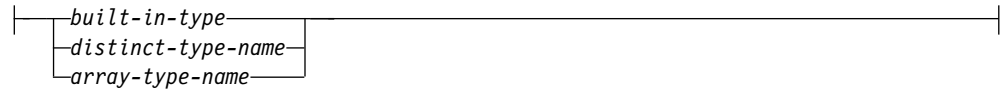
parameter-declaration:



parameter-type:

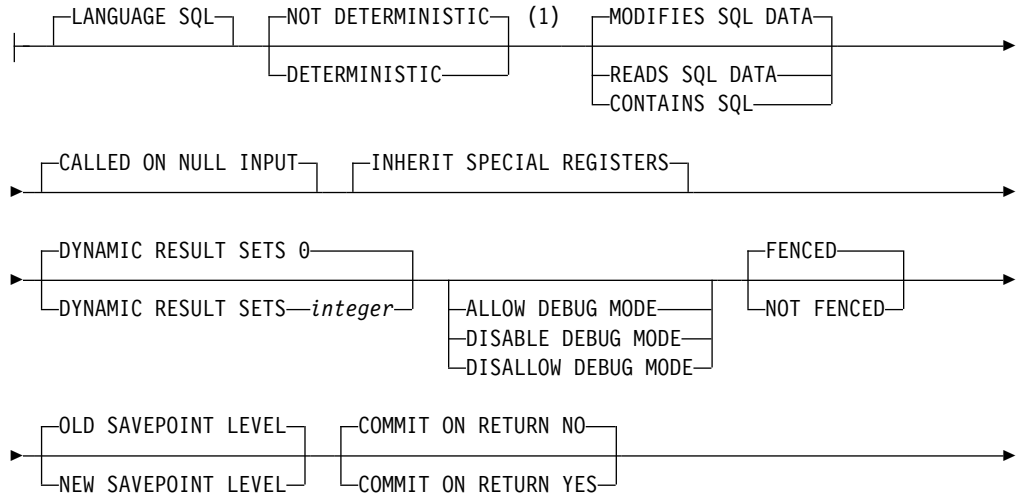


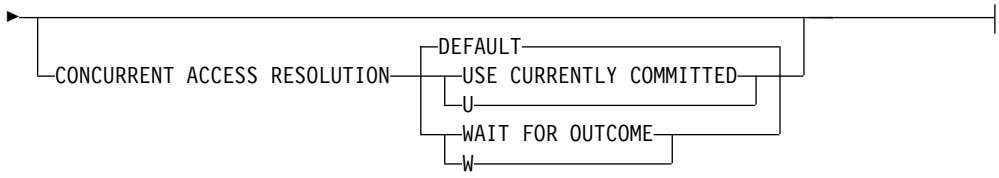
data-type:



default-clause:

option-list:





Notes:

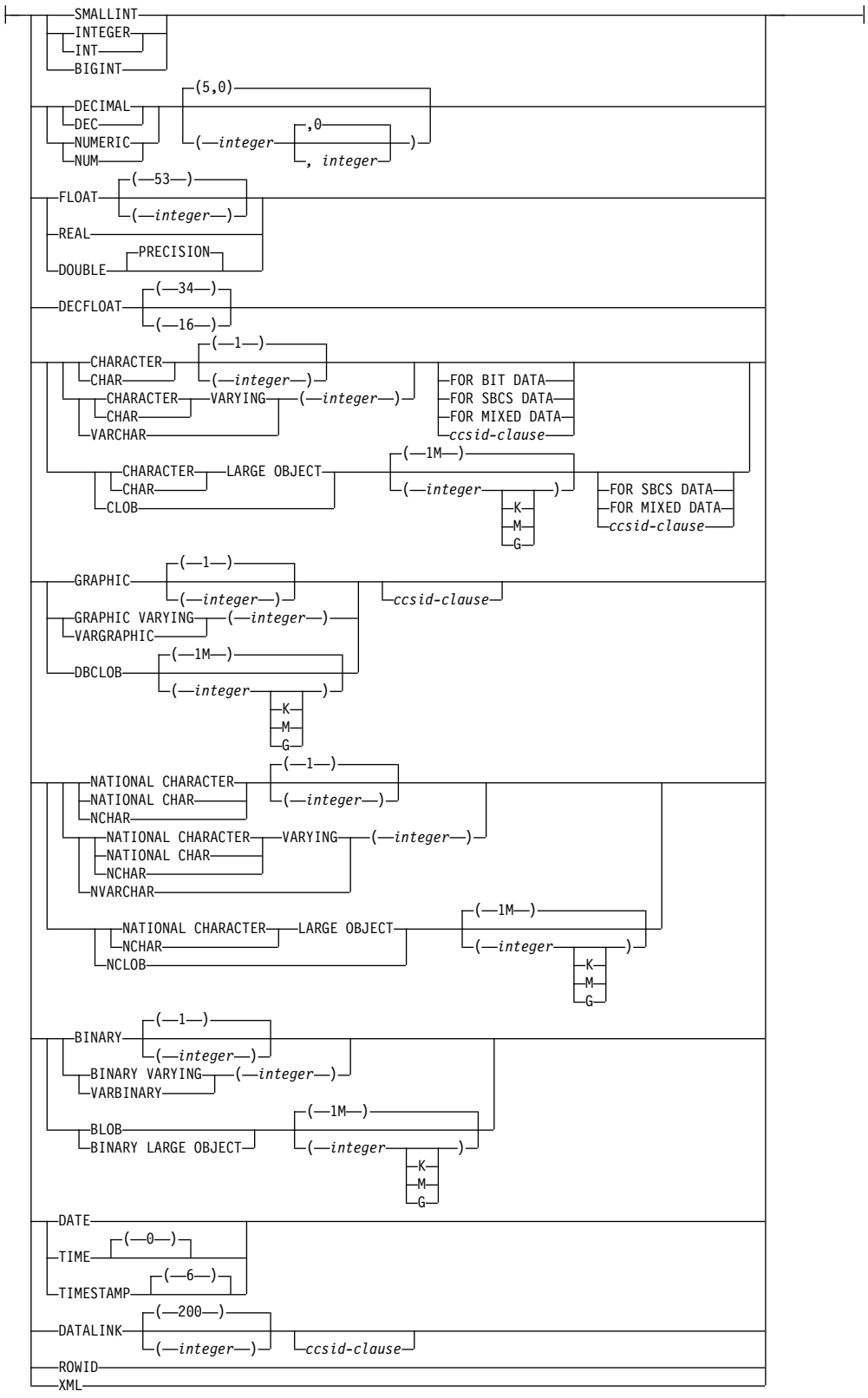
1 The clauses in the *option-list* can be specified in any order.

SQL-routine-body:

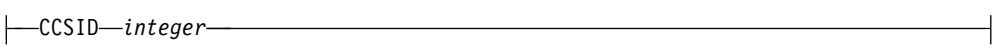

SQL-control-statement
ALLOCATE CURSOR-statement
ALLOCATE DESCRIPTOR-statement
ALTER FUNCTION-statement
ALTER PROCEDURE-statement
ALTER SEQUENCE-statement
ALTER TABLE-statement
ASSOCIATE LOCATORS-statement
COMMENT-statement
COMMIT-statement
CONNECT-statement
CREATE ALIAS-statement
CREATE FUNCTION (External Scalar)-statement
CREATE FUNCTION (External Table)-statement
CREATE FUNCTION (Sourced)-statement
CREATE INDEX-statement
CREATE PROCEDURE (External)-statement
CREATE SCHEMA-statement
CREATE SEQUENCE-statement
CREATE TABLE-statement
CREATE TYPE-statement
CREATE VIEW-statement
DEALLOCATE DESCRIPTOR-statement
DECLARE GLOBAL TEMPORARY TABLE-statement
DELETE-statement
DESCRIBE-statement
DESCRIBE CURSOR-statement
DESCRIBE INPUT-statement
DESCRIBE PROCEDURE-statement
DESCRIBE TABLE-statement
DISCONNECT-statement
DROP-statement
EXECUTE IMMEDIATE-statement
GET DESCRIPTOR-statement
GRANT-statement
INSERT-statement
LABEL-statement
LOCK TABLE-statement
MERGE-statement
REFRESH TABLE-statement
RELEASE-statement
RELEASE SAVEPOINT-statement
RENAME-statement
REVOKE-statement
ROLLBACK-statement
SAVEPOINT-statement
SELECT INTO-statement
SET CONNECTION-statement
SET CURRENT DEBUG MODE-statement
SET CURRENT DECFLOAT ROUNDING MODE-statement
SET CURRENT DEGREE-statement
SET CURRENT IMPLICIT XMLPARSE OPTION-statement
SET DESCRIPTOR-statement
SET ENCRYPTION PASSWORD-statement
SET PATH-statement
SET RESULT SETS-statement

SQL-routine-body (continued):

built-in-type:



ccsid-clause:



Description

PROCEDURE or SPECIFIC PROCEDURE

Identifies the procedure to alter. *procedure-name* must identify an SQL procedure that exists at the current server.

The specified procedure is altered. The owner of the procedure and all privileges on the procedure are preserved.

PROCEDURE *procedure-name*

Identifies the procedure by its name. The *procedure-name* must identify exactly one SQL procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type,...)*

Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type,...)* must identify an SQL procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types and the logical concatenation of the data types is used to identify the specific procedure instance which is being altered. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name()* is specified, the procedure identified must have zero parameters.

procedure-name

Identifies the name of the procedure.

(parameter-type,...)

Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parenthesis indicates that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its precision value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

ALTER *option-list*
Indicates that one or more of the options of the procedure are to be altered. If ALTER PROCEDURE ALTER *option-list* is specified and an option is not specified, the value from the existing procedure definition is used. See "CREATE PROCEDURE (SQL)" on page 955 for a description of each option.

REPLACE *routine-specification*
Indicates that the existing procedure definition, including options and parameters, is to be replaced by those specified in this statement. The values of all options are replaced when a procedure is replaced. If an option is not specified, the same default is used as when a new SQL procedure is created, for more information see "CREATE PROCEDURE (SQL)" on page 955.

If the routine has a comment or label, they are removed from the routine definition.

(*parameter-declaration***,...)**
Specifies the number of parameters of the procedure, the data type of each parameter, and the name of each parameter. A parameter for a procedure can be used for input only, for output only, or for both input and output.

The maximum number of parameters allowed in an SQL procedure is 1024.

IN Identifies the parameter as an input parameter to the procedure.

OUT
Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT
Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned. If an INOUT parameter is defined with a default and the default is used when calling the procedure, no value for the parameter is returned.

parameter-name
Names the parameter for use as an SQL variable. The name cannot be the same as any other *parameter-name* for the procedure.

data-type
Specifies the data type of the parameter. If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the procedure. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the procedure is called.

default-clause
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The expression is any expression defined in "Expressions" on page 165, that does not include an aggregate function or column name. If a default value is not specified, the parameter has no default and cannot be omitted on invocation. The maximum length of the expression string is 64K.

The default expression must not modify SQL data. The expression must be assignment compatible to the parameter data type. All objects referenced in a default expression must exist when the procedure is created.

Any comma in the default expression that is intended as a separator of numeric constants in a list must be followed by a space.

A default cannot be specified:

- for an OUT parameter.
- for a parameter of type array.

option-list
List of options for the procedure being altered. These options are the same ones that are listed above under ALTER *option-list*. If a specific option is not specified, the same default that is used when a new procedure is created is used. For more information see "CREATE PROCEDURE (SQL)" on page 955.

SET OPTION-statement
Specifies the options that will be used to create the procedure. For example, to create a debuggable procedure, the following statement could be included:

```
SET OPTION DBGVIEW = *SOURCE
```

For more information, see "SET OPTION" on page 1368.

The options CNULRQD, COMPILEOPT, NAMING, and SQLCA are not allowed in the ALTER PROCEDURE statement. The following options are used when processing default value expressions: ALWCPYDTA, CONACC, DATFMT, DATSEP, DECFLTRND, DECMPT, DECRESULT, DFTRDBCOL, LANGID, SQLCURRULE, SQLPATH, SRTSEQ, TGTRLS, TIMFMT, and TIMSEP.

SQL-routine-body
Specifies a single SQL statement, including a compound statement. See Chapter 7, "SQL control statements," on page 1427 for more information about defining SQL procedures.

CONNECT, SET CONNECTION, RELEASE, DISCONNECT, and SET TRANSACTION statements are not allowed in a procedure that is running on a remote application server. COMMIT and ROLLBACK statements are not allowed in an ATOMIC SQL procedure or in a procedure that is running on a connection to a remote application server.

ALTER PROCEDURE (SQL), ALTER FUNCTION (SQL Scalar), and ALTER FUNCTION (SQL Table) with a REPLACE keyword are not allowed in an *SQL-routine-body*.

Notes

General considerations for defining or replacing procedures: See CREATE PROCEDURE for general information about defining a procedure. ALTER PROCEDURE (SQL) allows individual attributes or the routine specification to be

altered while preserving the privileges on the procedure.

Alter Procedure Replace considerations: When an SQL procedure definition is replaced, SQL creates a temporary source file that will contain C source code with embedded SQL statements. A program object is then created using the CRTPGM command. The SQL options used to create the program are the options that are in effect at the time the ALTER PROCEDURE (SQL) statement is executed. The program is created with ACTGRP(*CALLER).

When an SQL procedure is altered, a new *PGM or *SRVPGM object is created and the procedure's attributes are stored in the created program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes.

The specific name is used as the name of the member in the source file and the name of the program object, if it is a valid system name. If the procedure name is not a valid system name, a unique name is generated. If a source file member with the same name already exists, the member is overlaid. If a module or a program with the same name already exists, the objects are not overlaid, and a unique name is generated. The unique names are generated according to the rules for generating system table names.

Target release considerations: When an SQL procedure definition is replaced, the target release will be the current release in which the ALTER statement is executed unless the user explicitly specifies a different target release. The target release can be explicitly specified using the TGTRLS keyword in the SET OPTION statement. If the ALTER is specified in the source for a RUNSQLSTM or CRTSQLxxx command, the TGTRLS keyword can also be specified on the command.

If the procedure definition is not replaced, the target release of the existing procedure will be preserved unless the target release level of the procedure is earlier than the earliest supported release level. In this case, the target release will be changed to the earliest supported release level.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL can be used as synonyms for CALLED ON NULL INPUT.
- DYNAMIC RESULT SET, RESULT SETS, and RESULT SET may be used as synonyms for DYNAMIC RESULT SETS.

Examples

Modify the definition for an SQL procedure so that SQL changes are committed on return from the SQL procedure.

```
ALTER PROCEDURE UPDATE_SALARY_2
  ALTER COMMIT ON RETURN YES
```

ALTER SEQUENCE

The ALTER SEQUENCE statement can be used to change a sequence.

The ALTER SEQUENCE statement can be used to change a sequence in any of these ways:

- Restarting the sequence
- Changing the increment between future sequence values
- Setting or eliminating the minimum or maximum values
- Changing the number of cached sequence numbers
- Changing the attribute that determines whether the sequence can cycle or not
- Changing whether sequence numbers must be generated in order of request

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the sequence identified in the statement:
  - The system authority *EXECUTE on the library containing the sequence
  - The ALTER privilege for the sequence
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - *USE to the Change Data Area (CHGDTAARA) command
  - *USE to the Retrieve Data Area (RTVDTAARA) command
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSSEQOBJECTS catalog table:
  - The UPDATE privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

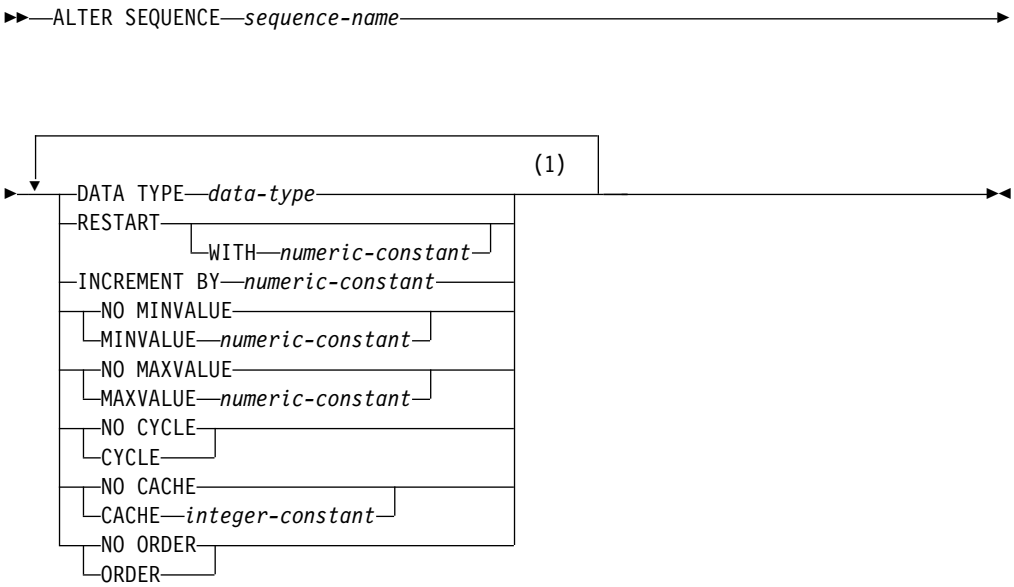
If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

For information on the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Sequence,

Corresponding System Authorities When Checking Privileges to a Table or View, and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax

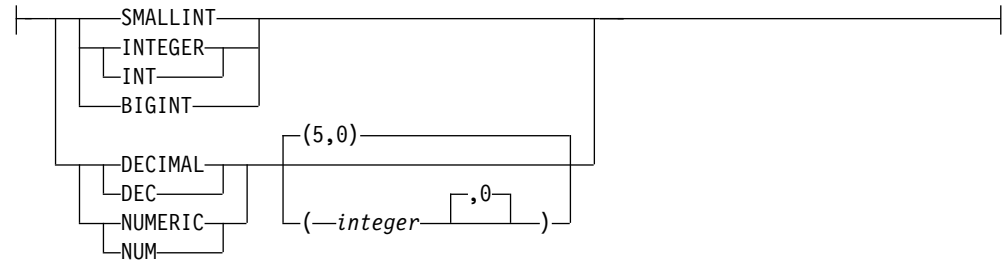


Notes:

1 The same clause must not be specified more than once.

data-type:

built-in-type:



Description

sequence-name
Identifies the sequence to be altered. The name must identify a sequence that already exists at the current server.

DATA TYPE *data-type*
Specifies the new data type to be used for the sequence value. The data type can be any exact numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or

NUMERIC) with a scale of zero, or a user-defined distinct type for which the source type is an exact numeric type with a scale of zero.

Each of the existing START WITH, INCREMENT BY, MINVALUE, and MAXVALUE attributes that are not changed by the ALTER SEQUENCE statement must contain a value that could be assigned to a column of the data type associated with the new data type.

built-in-type
Specifies the new built-in data type used as the basis for the internal representation of the sequence. If the data type is DECIMAL or NUMERIC, the precision must be less than or equal to 63 and the scale must be 0. See "CREATE TABLE" on page 982 for a more complete description of each built-in data type.

For portability of applications across platforms, use DECIMAL instead of a NUMERIC data type.

distinct-type-name
Specifies that the new data type of the sequence is a distinct type (a user-defined data type). If the source type is DECIMAL or NUMERIC, the precision of the sequence is the precision of the source type of the distinct type. The precision of the source type must be less than or equal to 63 and the scale must be 0. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path.

RESTART
Restarts the sequence. If *numeric-constant* is not specified, the sequence is restarted at the value specified implicitly or explicitly as the starting value on the CREATE SEQUENCE statement that originally created the sequence.

WITH *numeric-constant*
Restarts the sequence with the specified value. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence, without nonzero digits to the right of the decimal point.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the sequence. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence, and does not exceed the value of a large integer constant, without nonzero digits existing to the right of the decimal point.

If this value is negative, then this is a descending sequence. If this value is 0 or positive, this is an ascending sequence after the ALTER statement.

NO MINVALUE or MINVALUE
Specifies the minimum value at which a descending sequence either cycles or stops generating values, or an ascending sequence cycles to after reaching the maximum value.

NO MINVALUE
For an ascending sequence, the value is the original starting value. For a descending sequence, the value is the minimum value of the data type (and precision, if DECIMAL or NUMERIC) associated with the sequence.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value that is generated for this sequence. This value can be any positive or negative value that could be assigned to a column of the data type associated with the

sequence and without non-zero digits to the right of the decimal point. The value must be less than or equal to the maximum value.

NO MAXVALUE or MAXVALUE
Specifies the maximum value at which an ascending sequence either cycles or stops generating values, or a descending sequence cycles to after reaching the minimum value.

NO MAXVALUE
For an ascending sequence, the value is the maximum value of the data type (and precision, if DECIMAL or NUMERIC) associated with the sequence. For a descending sequence, the value is the original starting value.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value that is generated for this sequence. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence and without non-zero digits to the right of the decimal point. The value must be greater than or equal to the minimum value.

CYCLE or NO CYCLE
Specifies whether this sequence should continue to generate values after reaching either the maximum or minimum value of the sequence.

NO CYCLE
Specifies that values will not be generated for the sequence once the maximum or minimum value for the sequence has been reached.

CYCLE
Specifies that values continue to be generated for this sequence after the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches the maximum value of the sequence, it generates its minimum value. After a descending sequence reaches its minimum value of the sequence, it generates its maximum value. The maximum and minimum values for the sequence determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated for a sequence by the database manager.

CACHE or NO CACHE
Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of the NEXT VALUE sequence expression.

CACHE *integer-constant*
Specifies the maximum number of sequence values that are preallocated and kept in memory. Preallocating and storing values in the cache reduces synchronous I/O when values are generated for the sequence.

In certain situations, such as system failure, all cached sequence values that have not been used in committed statements are lost, and thus, will never be used. The value specified for the CACHE option is the maximum number of sequence values that could be lost in these situations.

The minimum value is 2.

NO CACHE
Specifies that values of the sequence are not to be preallocated. It ensures that there is not a loss of values in situations, such as system failure. When

this option is specified, the values of the sequence are not stored in the cache. In this case, every request for a new value for the sequence results in synchronous I/O.

NO ORDER or ORDER
Specifies whether the sequence numbers must be generated in order of request.

NO ORDER
Specifies that the sequence numbers do not need to be generated in order of request.

ORDER
Specifies that the sequence numbers are generated in order of request. If ORDER is specified, the performance of the NEXT VALUE sequence expression will be worse than if NO ORDER is specified.

Notes

Altering a sequence:

- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- All the cached values are lost when a sequence is altered.
- After restarting a sequence or changing it to cycle, it is possible that a generated value will duplicate a value previously generated for that sequence.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases of other DB2 products. These keywords are non-standard and should not be used:

- The keywords NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER can be used as synonyms for NO MINVALUE, NO MAXVALUE, NO CYCLE, NO CACHE, and NO ORDER.

Examples

A possible reason for specifying RESTART without a numeric value would be to reset the sequence to the START WITH value. In this example, the goal is to generate the numbers from 1 up to the number of rows in a table and then inserting the numbers into a column added to the table using temporary tables.

```
ALTER SEQUENCE ORG_SEQ RESTART

DECLARE GLOBAL TEMPORARY TABLE TEMP_ORG AS
  (SELECT NEXT VALUE FOR ORG_SEQ, ORG.*
  FROM ORG) WITH DATA

INSERT INTO TEMP_ORG
  SELECT NEXT VALUE FOR ORG_SEQ, ORG.*
  FROM ORG
```

Another use would be to get results back where all the resulting rows are numbered:

```
ALTER SEQUENCE ORG_SEQ RESTART

SELECT NEXT VALUE FOR ORG_SEQ, ORG.*
  FROM ORG
```

ALTER TABLE

The ALTER TABLE statement alters the definition of a table.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table identified in the statement,
  - The ALTER privilege on the table, and
  - The system authority *EXECUTE on the library containing the table
- Administrative authority

To define a foreign key, the privileges held by the authorization ID of the statement must include at least one of the following on the parent table:

- The REFERENCES privilege or object management authority for the table
- The REFERENCES privilege on each column of the specified parent key
- Administrative authority

If a field procedure is defined, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority *EXECUTE on the program, and
  - The system authority *EXECUTE on the library containing the program
- Administrative authority

If a *select-statement* is specified, the privileges held by the authorization ID of the statement must include at least one of the following on the tables or views specified in these clauses:

- The SELECT privilege for the table or view
- Administrative authority

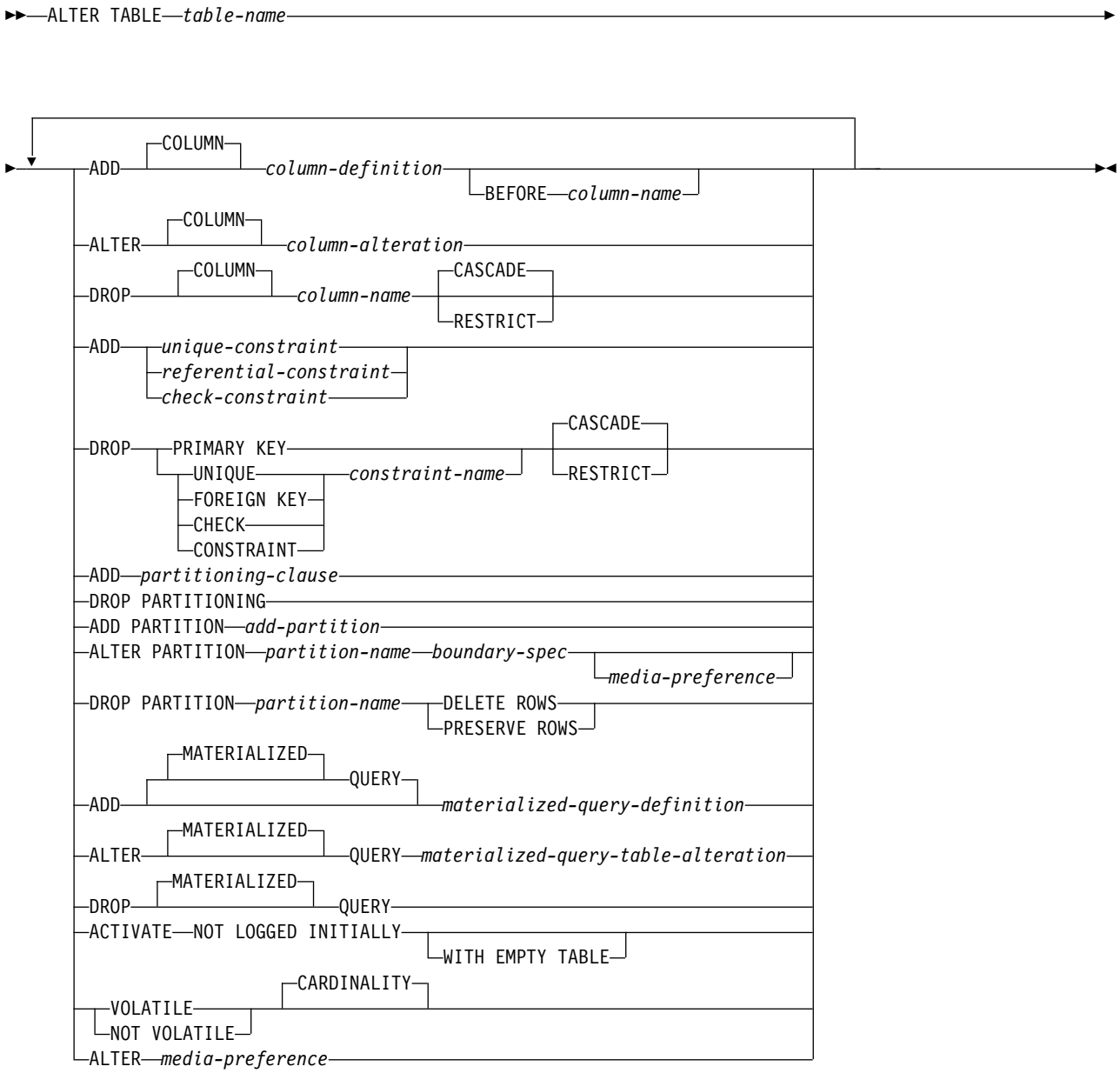
If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

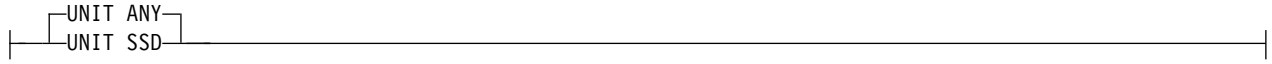
For information on the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View and Corresponding System Authorities When Checking Privileges to a Distinct Type.

ALTER TABLE

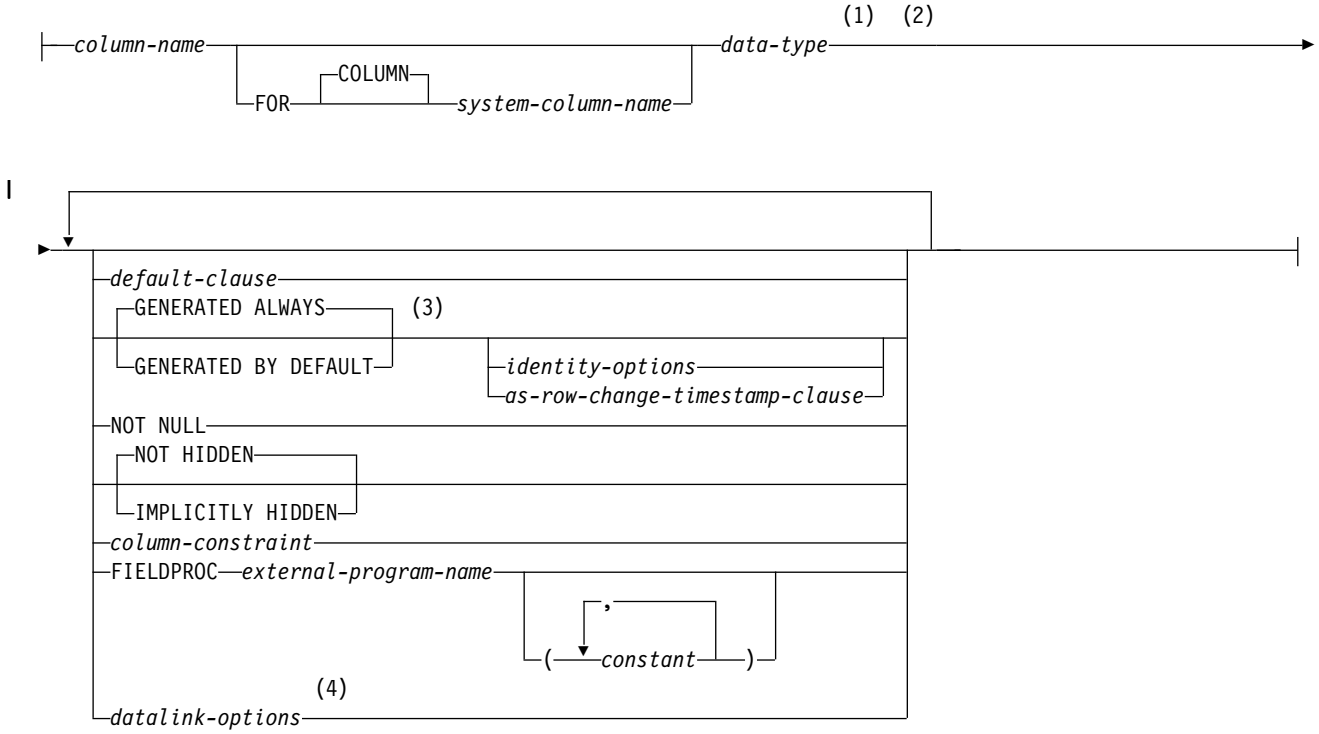
Syntax



media-preference:



column-definition:



data-type:

Notes:

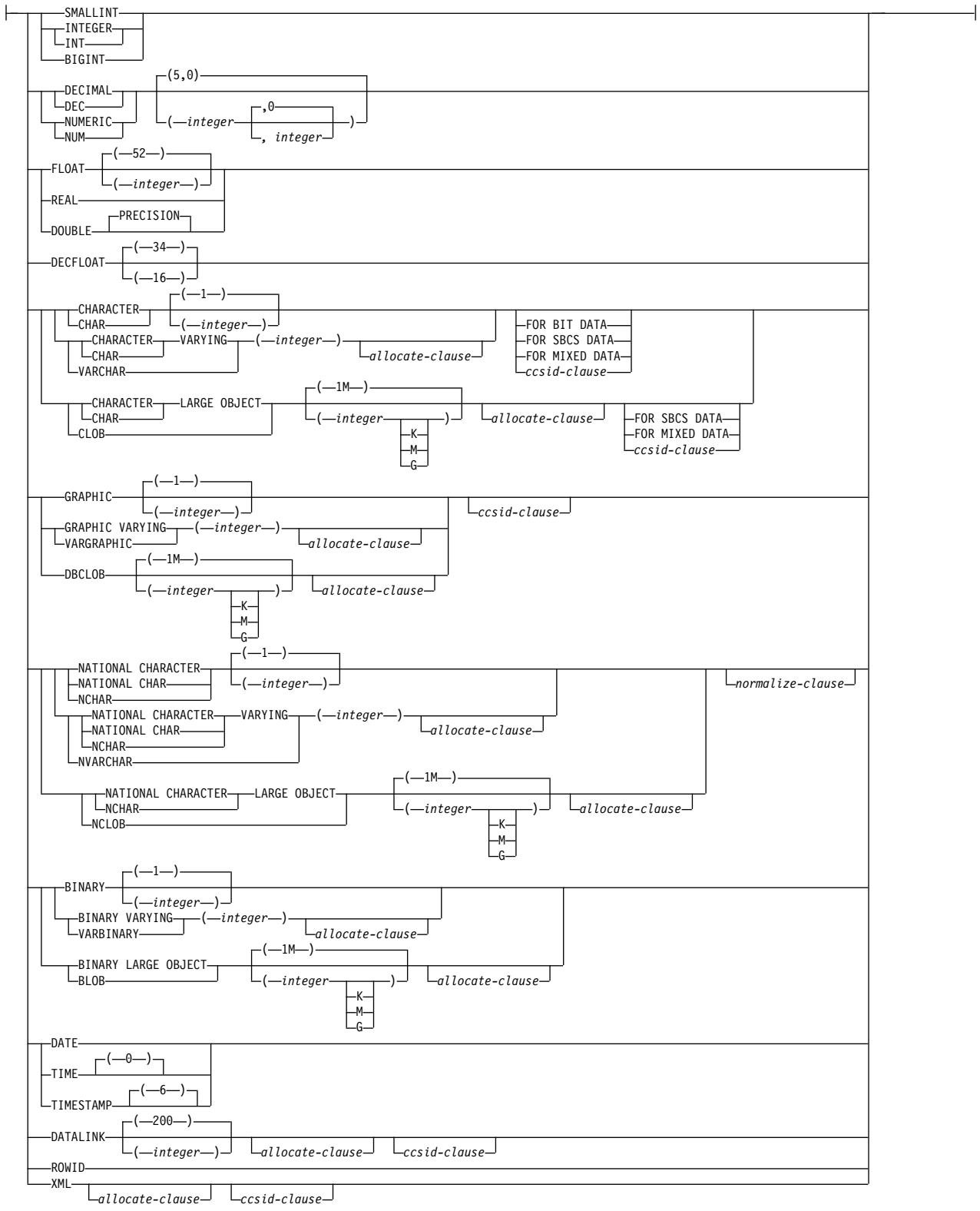
1 *data-type* is optional for row change timestamp columns

2 The same clause must not be specified more than once.

3 GENERATED can be specified only if the column has a ROWID data type (or a distinct type that is based on a ROWID data type), the column is an identity column, or the column is a row change timestamp.

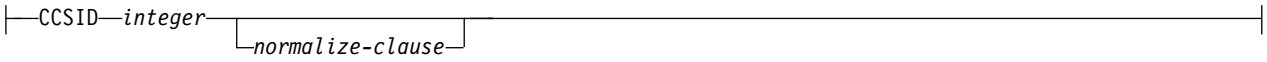
4 The *datalink-options* can only be specified for DATALINKs and distinct types sourced on DATALINKs.

built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(integer
,0
, integer
)
FLOAT
(52)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
allocate-clause
FOR BIT DATA
FOR SBCS DATA
FOR MIXED DATA
ccsid-clause
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
K
M
G
GRAPHIC
GRAPHIC VARYING
VARGRAPHIC
DBCLOB
NATIONAL CHARACTER
NATIONAL CHAR
NCHAR
NATIONAL CHARACTER VARYING
NATIONAL CHAR VARYING
NVARCHAR
NCLOB
normalize-clause
BINARY
BINARY VARYING
VARBINARY
BINARY LARGE OBJECT
BLOB
DATE
TIME
(0)
TIMESTAMP
(6)
DATALINK
(200)
ROWID
XML

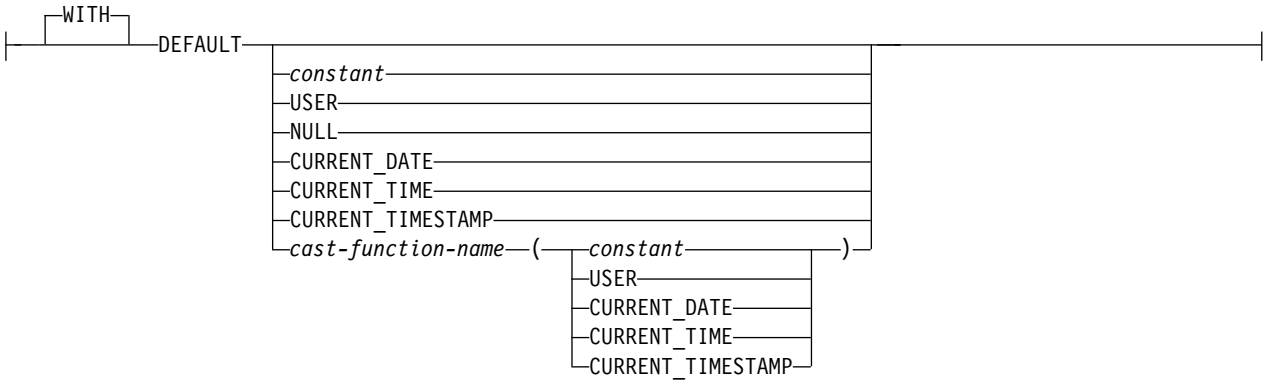
allocate-clause:

ccsid-clause:

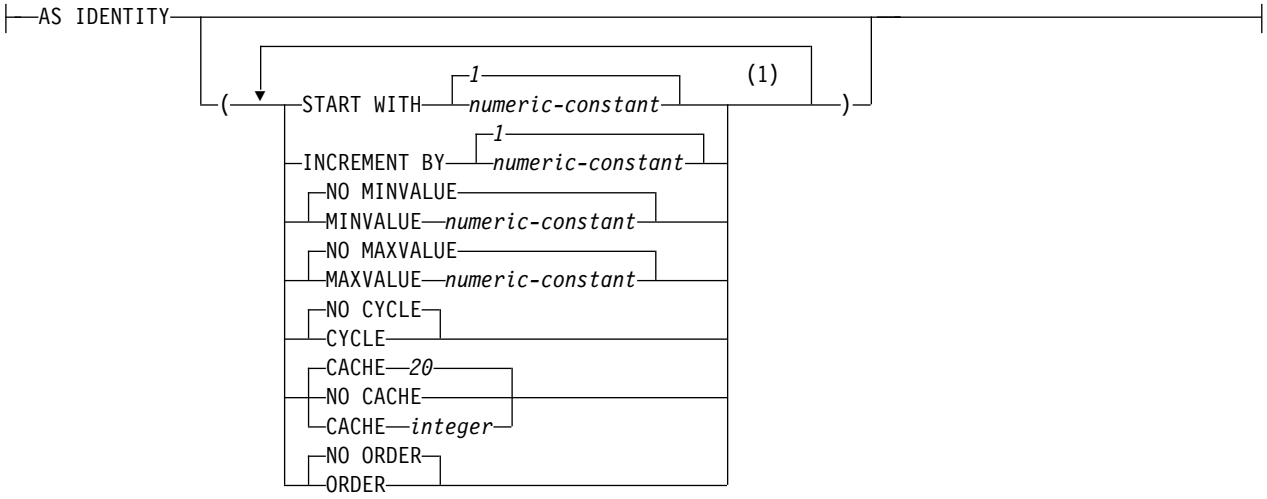


normalize-clause:

default-clause:



identity-options:

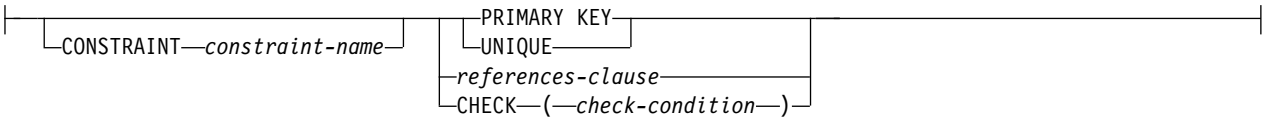


Notes:

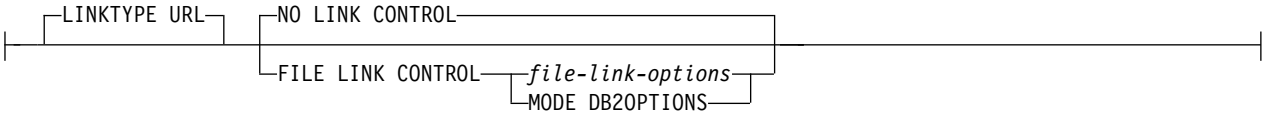
1 The same clause must not be specified more than once.

as-row-change-timestamp-clause:

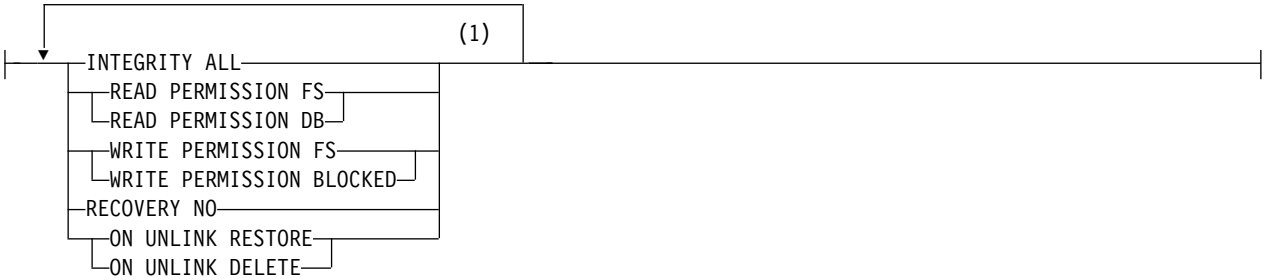
column-constraint:



datalink-options:



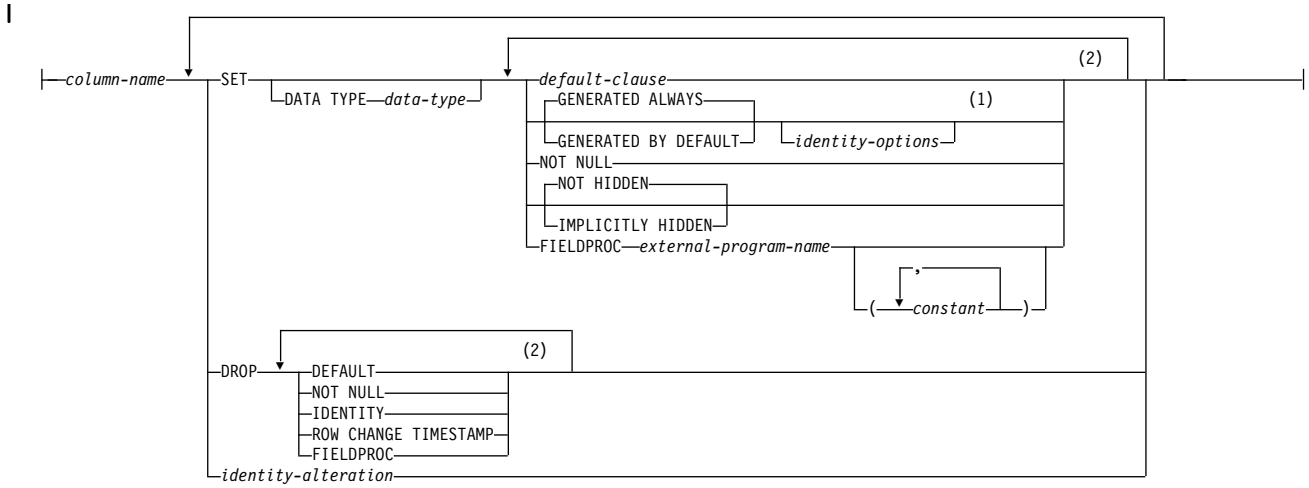
file-link-options:



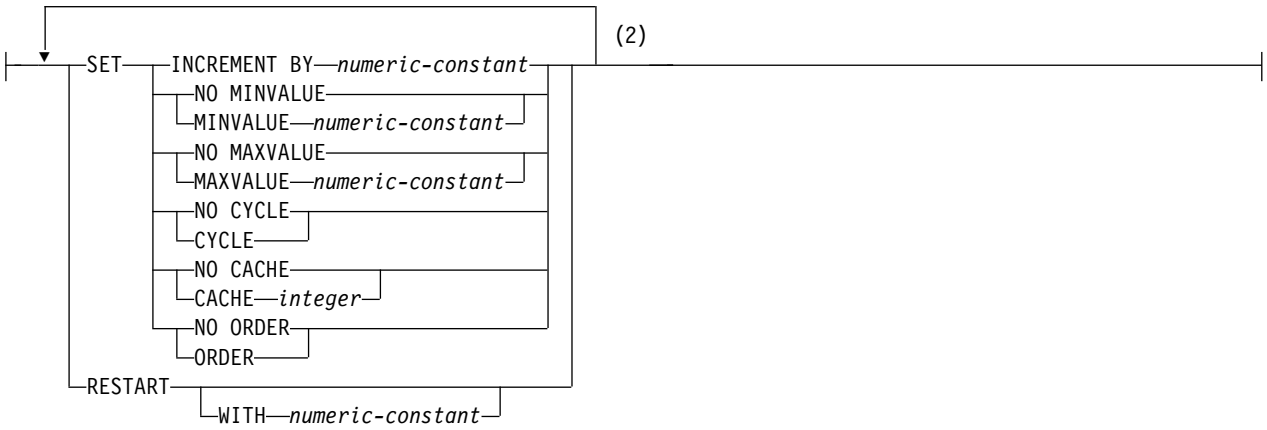
Notes:

1 All five *file-link-options* must be specified, but they can be specified in any order.

column-alteration:



identity-alteration:

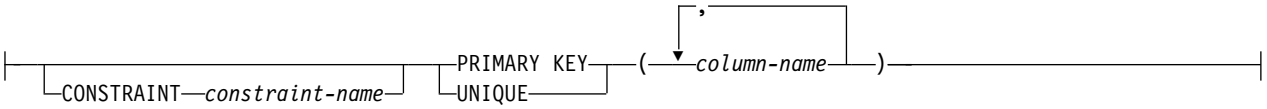


Notes:

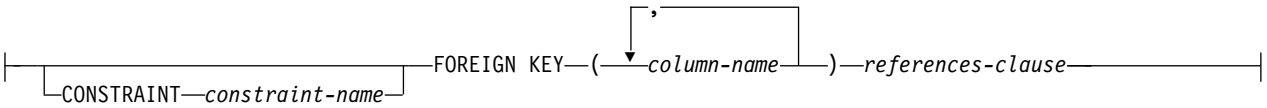
1 GENERATED can be specified only if the column has a ROWID data type (or a distinct type that is based on a ROWID data type), the column is an identity column, *identity-options* are specified, or the column is a row change timestamp.

2 The same clause must not be specified more than once.

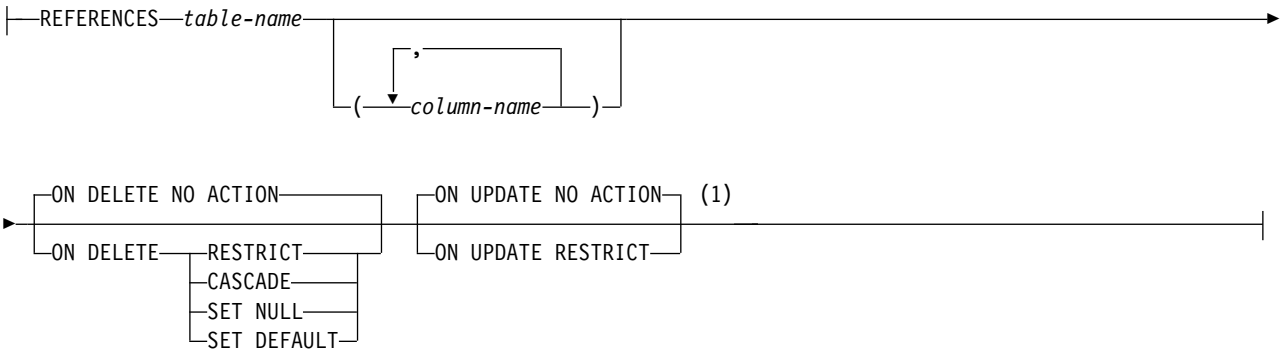
unique-constraint:



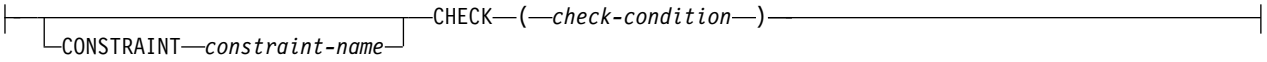
referential-constraint:



references-clause:



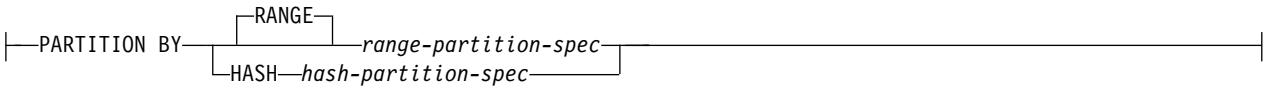
check-constraint:



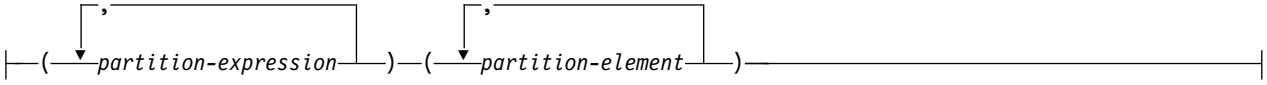
Notes:

1 The ON DELETE and ON UPDATE clauses may be specified in either order.

partitioning-clause:

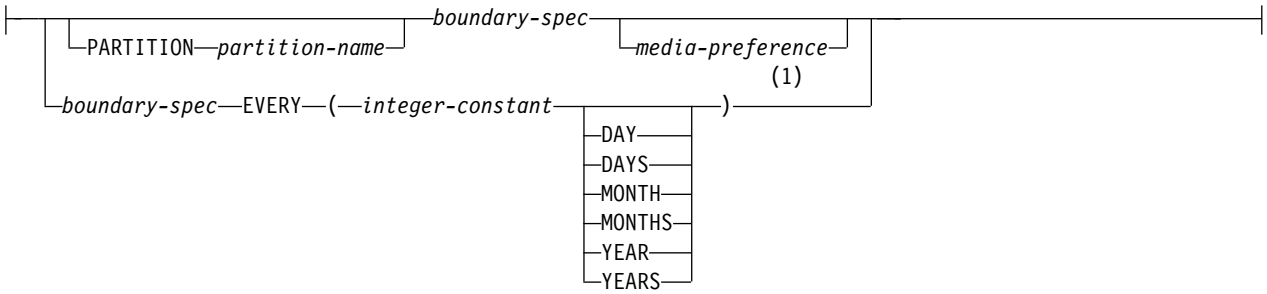


range-partition-spec:

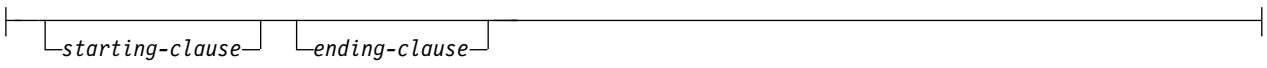


partition-expression:

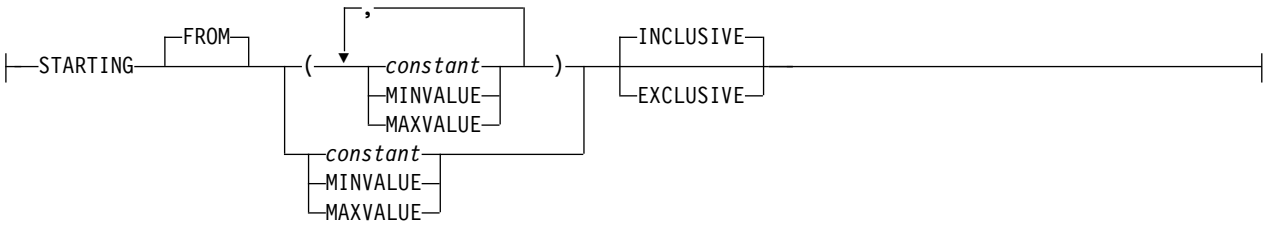
partition-element:



boundary-spec:



starting-clause:

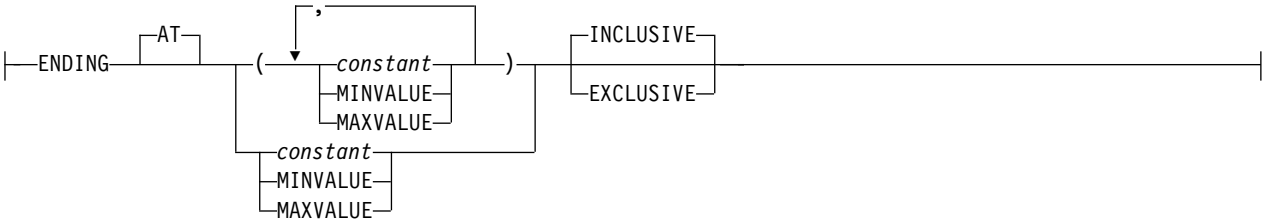


Notes:

1 This syntax for a *partition-element* is valid if there is only one *partition-expression* with a numeric or datetime data type.

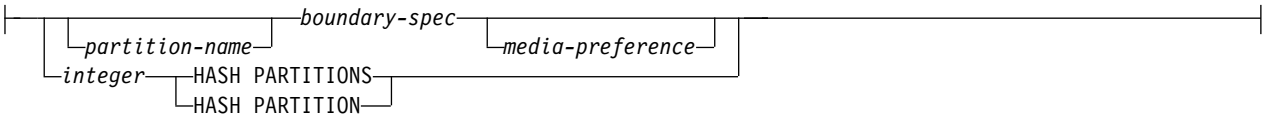
ALTER TABLE

ending-clause:

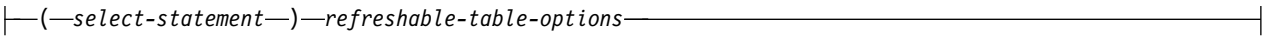


hash-partition-spec:

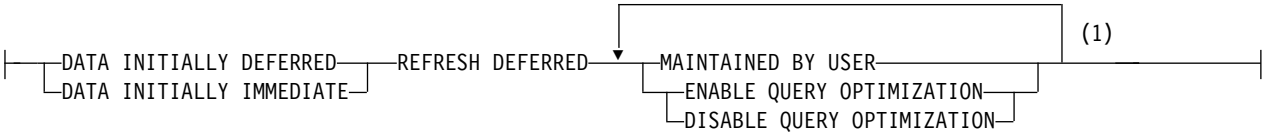
add-partition:



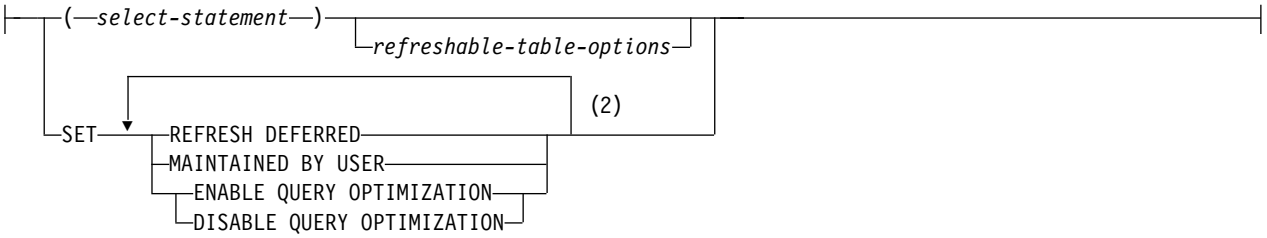
materialized-query-definition:



refreshable-table-options:



materialized-query-table-alteration:



Notes:

1 The same clause must not be specified more than once. MAINTAINED BY USER must be specified.

2 The same clause must not be specified more than once.

Description

table-name
Identifies the table to be altered. The *table-name* must identify a table that exists at the current server. It must not be a view, a catalog table, or a declared temporary table. If *table-name* identifies a materialized query table, ADD *column-definition*, ALTER *column-alteration*, and DROP COLUMN are not allowed.

ADD COLUMN *column-definition*

Adds a column to the table. If the table has rows, every value of the column is set to its default value, unless the column is a ROWID column, an identity column (a column that is defined AS IDENTITY), or a row change timestamp. The database manager generates default values for ROWID columns and identity columns. If the table previously had *n* columns, the ordinality of the new column is *n*+1. The value of *n*+1 must not exceed 8000.

A table can have only one ROWID, identity column, or row change timestamp.

A DataLink column with FILE LINK CONTROL cannot be added to a table that is a dependent in a referential constraint with a delete rule of CASCADE.

Adding a new column must not make the sum of the row buffer byte counts of the columns be greater than 32766 or, if a VARCHAR, VARBINARY, or VARGRAPHIC column is specified, 32740. Additionally, if a LOB or XML column is specified, the sum of the byte counts of the columns must not be greater than 3 758 096 383 at the time of insert or update. For information on the byte counts of columns according to data type, see "Maximum row sizes" on page 1027.

column-name
Names the column to be added to the table. Do not use the same name for more than one column of the table or for a *system-column-name* of the table. Do not qualify *column-name*.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. Do not use the same name for more than one *column-name* or *system-column-name* of the table.

If the *system-column-name* is not specified, and the *column-name* is not a valid *system-column-name*, a system column name is generated. For more information about how system column names are generated, see "Rules for Column Name Generation" on page 1029.

data-type
Specifies the data type of the column. The data type can be a built-in data type or a distinct type.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 982 for a description of built-in types.

distinct-type-name
Specifies that the data type of a column is a distinct type. The length, precision and scale of the column are respectively the length, precision, and scale of the source type of the distinct type. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path.

DEFAULT

Specifies a default value for the column. This clause cannot be specified more than once in the same *column-definition*. DEFAULT cannot be specified for an XML column, a ROWID column, an identity column (a column that is defined AS IDENTITY), or a row change timestamp column. The database manager generates default values for ROWID columns, identity columns, and row change timestamp columns. If a value is not specified following the DEFAULT keyword or a DEFAULT clause is not specified, then:

- if the column is nullable, the default value is the null value.
- if the column is not nullable, the default depends on the data type of the column:

Data type	Default value
Numeric	0
Fixed-length character or graphic string	Blanks
Fixed-length binary string	Hexadecimal zeros
Varying-length string	A string length of 0
Date	For existing rows, a date corresponding to 1 January 0001. For added rows, the current date.
Time	For existing rows, a time corresponding to 0 hours, 0 minutes, and 0 seconds. For added rows, the current time.
Timestamp	For existing rows, • If the column is a row change timestamp, the current timestamp. • Otherwise, a date corresponding to 1 January 0001 and a time corresponding to 0 hours, 0 minutes, 0 seconds, and 0 microseconds. For added rows, the current timestamp.
Datalink	A value corresponding to DLVALUE('','URL','')
distinct-type	The default value of the corresponding source type of the distinct type.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

constant

Specifies the constant as the default for the column. The specified constant must represent a value that could be assigned to the column in accordance with the rules of assignment as described in "Assignments and comparisons" on page 98. A floating-point constant or decimal floating-point constant must not be used for a SMALLINT, INTEGER, BIGINT, DECIMAL, or NUMERIC column. A decimal constant must not contain more digits to the right of the decimal point than the specified scale of the column.

USER

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value for the column. The data type of the column must be CHAR or VARCHAR with a length attribute that is greater than

or equal to the length attribute of the USER special register. For existing rows, the value is that of the USER special register at the time the ALTER TABLE statement is processed.

NULL
Specifies null as the default for the column. If NOT NULL is specified, DEFAULT NULL must not be specified within the same *column-definition*.

CURRENT_DATE
Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the column must be DATE or a distinct type based on a DATE.

CURRENT_TIME
Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the column must be TIME or a distinct type based on a TIME.

CURRENT_TIMESTAMP
Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the column must be TIMESTAMP or a distinct type based on a TIMESTAMP.

cast-function-name
This form of a default value can only be used with columns defined as a distinct type, BINARY, VARBINARY, BLOB, CLOB, DBCLOB, DATE, TIME, or TIMESTAMP data types. The following table describes the allowed uses of these *cast-functions*.

Data Type	Cast Function Name
Distinct type N based on a BINARY, VARBINARY, BLOB, CLOB, or DBCLOB	BINARY, VARBINARY, BLOB, CLOB, or DBCLOB *
Distinct type N based on a DATE, TIME, or TIMESTAMP	N (the user-defined cast function that was generated when N was created) ** or DATE, TIME, or TIMESTAMP *
Distinct type N based on other data types	N (the user-defined cast function that was generated when N was created) **
BINARY, VARBINARY, BLOB, CLOB, or DBCLOB	BINARY, VARBINARY, BLOB, CLOB, or DBCLOB *
DATE, TIME, or TIMESTAMP	DATE, TIME, or TIMESTAMP *
Notes: * The name of the function must match the name of the data type (or the source type of the distinct type) with an implicit or explicit schema name of QSYS2. ** The name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type.	

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the

data type if not a distinct type. For BINARY, VARBINARY, BLOB, CLOB, DBCLOB, DATE, TIME, and TIMESTAMP functions, the constant must be a string constant.

USER
Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value for the column. The data type of the source type of the distinct type of the column must be CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of USER. For existing rows, the value is that of the USER special register at the time the ALTER TABLE statement is processed.

CURRENT_DATE
Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the source type of the distinct type of the column must be DATE.

CURRENT_TIME
Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the source type of the distinct type of the column must be TIME.

CURRENT_TIMESTAMP
Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the source type of the distinct type of the column must be TIMESTAMP.

If the value specified is not valid, an error is returned.

GENERATED
Specifies that the database manager generates values for the column. GENERATED may be specified if the column is to be considered an identity column (defined with the AS IDENTITY clause) or a row change timestamp column. It may also be specified if the data type of the column is a ROWID (or a distinct type that is based on a ROWID). Otherwise, it must not be specified.

ALWAYS
Specifies that the database manager will always generate a value for the column when a row is inserted or updated and a default value must be generated. ALWAYS is the recommended value.

BY DEFAULT
Specifies that the database manager will generate a value for the column when a row is inserted or updated and a default value must be generated, unless an explicit value is specified.

For a ROWID column, the database manager uses a specified value, but it must be a valid unique row ID value that was previously generated by the database manager or DB2 for i.

For an identity column or row change timestamp column, the database manager inserts or updates a specified value but does not verify that it is a unique value for the column unless the identity column or row change timestamp column has a unique constraint or a unique index that solely specifies the identity column or row change timestamp column.

AS IDENTITY
Specifies that the column is an identity column for the table. A table can have only one identity column. An identity column is not allowed in a distributed table. AS IDENTITY can be specified only if the data type for the column is an

exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL or NUMERIC with a scale of zero, or a distinct type based on one of these data types). If a DECIMAL or NUMERIC data type is specified, the precision must not be greater than 31.

An identity column is implicitly NOT NULL. See the AS IDENTITY clause in "CREATE TABLE" on page 982 for the descriptions of the identity attributes.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP
Specifies that the column is a timestamp and the values will be generated by the database manager. The database manager generates a value for the column for each row as a row is inserted, and for every row in which any column is updated. The value generated for a row change timestamp column is a timestamp corresponding to the time of the insert or update of the row. If multiple rows are inserted with a single SQL statement, the value for the row change timestamp column may be different for each row to reflect when each row was inserted. The generated value is not guaranteed to be unique.

A table can have only one row change timestamp column. If *data-type* is specified, it must be a TIMESTAMP or a distinct type based on a TIMESTAMP. A row change timestamp column cannot have a DEFAULT clause and must be NOT NULL.

NOT NULL
Prevents the column from containing null values. Omission of NOT NULL implies that the column can contain null values. If NOT NULL is specified in the column definition, then DEFAULT must also be specified unless the column is an identity column. NOT NULL is required for a row change timestamp column

NOT HIDDEN
Indicates the column is included in implicit references to the table in SQL statements. This is the default.

IMPLICITLY HIDDEN
Indicates the column is not visible in SQL statements unless it is referred to explicitly by name. For example, SELECT * does not include any hidden columns in the result. A table must contain at least one column that is not IMPLICITLY HIDDEN.

column-constraint
The *column-constraint* of a *column-definition* provides a shorthand method of defining a constraint composed of a single column. Thus, if a *column-constraint* is specified in the definition of column C, the effect is the same as if that constraint were specified as a *unique-constraint*, *referential-constraint* or *check-constraint* in which C is the only identified column.

CONSTRAINT *constraint-name*
A *constraint-name* must not be the same as a constraint name that was previously specified in the ALTER TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

PRIMARY KEY
Provides a shorthand method of defining a primary key composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

This clause must not be specified in more than one *column-definition* and must not be specified at all if the UNIQUE clause is specified in the column definition. The column must not be a LOB, DataLink, or XML column. If a sort sequence is specified, the column must not contain a field procedure.

When a primary key is added, a CHECK constraint is implicitly added to enforce the rule that the NULL value is not allowed in the column that makes up the primary key.

UNIQUE
Provides a shorthand method of defining a unique constraint composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE (C) clause is specified as a separate clause.

This clause cannot be specified more than once in a column definition and must not be specified if PRIMARY KEY is specified in the *column-definition*. The column must not be a LOB, DataLink, or XML column. If a sort sequence is specified, the column must not contain a field procedure.

references-clause
The *references-clause* of a column-definition provides a shorthand method of defining a foreign key composed of a single column. Thus, if a *references-clause* is specified in the definition of column C, the effect is the same as if that references-clause were specified as part of a FOREIGN KEY clause in which C is the only identified column. The *references-clause* is not allowed if the table is a declared global temporary table or a distributed table. The column cannot be a row change timestamp column.

CHECK(*check-condition***)**
Provides a shorthand method of defining a check constraint whose *check-condition* only references a single column. Thus, if CHECK is specified in the column definition of column C, no columns other than C can be referenced in the *check-condition* of the check constraint. The effect is the same as if the check constraint were specified as a separate clause.

ROWID, XML, or DATALINK with FILE LINK CONTROL columns cannot be referenced in a CHECK constraint. For additional restrictions see, "ADD check-constraint" on page 781.

FIELDPROC
Designates an *external-program-name* as the field procedure exit routine for the column. It must be an ILE program that does not contain SQL. It cannot be a service program.

The field procedure encodes and decodes column values. Before a value is inserted in the column, it is passed to the field procedure for encoding. Before a value from the column is used, it is passed to the field procedure for decoding.

The field procedure is also invoked during the processing of the ALTER TABLE statement. When so invoked, the procedure provides DB2 with the column's field description. The field description defines the data characteristics of the encoded values. By contrast, the information supplied for the column defines the data characteristics of the decoded values.

constant
Specifies a parameter that is passed to the field procedure when it is invoked. A parameter list is optional.

A field procedure cannot be defined for a column that is a ROWID or DATALINK or a distinct type based on a ROWID or DATALINK. The column must not be an identity column or a row change timestamp column. The column must not have a default value of CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP, or USER. The column cannot be referenced in a check condition, unless it is referenced in a NULL predicate. If it is part of a foreign key, the corresponding parent key column must use the same field procedure. See SQL Programming for more details on how to create a field procedure.

datalink-options
Specifies the options associated with a DATALINK column. See "CREATE TABLE" on page 982 for a description of *datalink-options*.

BEFORE *column-name*
Identifies the column before which the new column is added. The name must not be qualified and must identify an existing column in the table. If the BEFORE clause is not specified, the column is added at the end of the row.

ALTER COLUMN *column-alteration*

Alters the definition of a column, including the attributes of an existing identity column. Only the attributes specified will be altered. Others will remain unchanged.

column-name
Identifies the column to be altered. The name must not be qualified and must identify an existing column in the table. The name must not identify a column that is being added or dropped in the same ALTER TABLE statement.

SET DATA TYPE *data-type*
Specifies the new data type of the column to be altered. The new data type must be compatible with the existing data type of the column. For more information about the compatibility of data types see "Assignments and comparisons" on page 98. However, changing a datetime data type to a character-string data type or a numeric data type to a character-string data type or a character-string data type to a numeric data type is not allowed. For an XML column, only the CCSID can be changed.

The specified length, precision, and scale may be larger, smaller, or the same as the existing length, precision, and scale. However, if the new length, precision, or scale is smaller, truncation or numeric conversion errors may occur.

If the specified column has a default value and a new default value is not specified, the existing default value must represent a value that could be assigned to the column in accordance with the rules for assignment as described in "Assignments and comparisons" on page 98.

If the column is specified in a unique, primary, or foreign key, the new sum of the lengths of the columns of the keys must not exceed 32766-*n*, where *n* is the number of columns specified that allow nulls.

Changing the attributes will cause any existing values in the column to be converted to the new column attributes according to the rules for assignment to a column, except that string values will be truncated.

SET *default-clause*
Specifies the new default value of the column to be altered. The specified default value must represent a value that could be assigned to the column in accordance with the rules for assignment as described in "Assignments and comparisons" on page 98.

SET GENERATED ALWAYS or GENERATED BY DEFAULT
Specifies that the database manager generates values for the column. GENERATED may be specified if the column is to be considered an identity column (defined with the AS IDENTITY clause), row change timestamp column, or the data type of the column is a ROWID (or a distinct type that is based on a ROWID). Otherwise, it must not be specified.

AS IDENTITY
Specifies that the column is changed to an identity column for the table. A table can have only one identity column. An identity column is not allowed in a distributed table. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL or NUMERIC with a scale of zero, or a distinct type based on one of these data types). If a DECIMAL or NUMERIC data type is specified, the precision must not be greater than 31.

The identity column is implicitly changed to NOT NULL. If the column has an explicit default value, the default value is dropped. See the AS IDENTITY clause in "CREATE TABLE" on page 982 for the descriptions of the identity attributes.

SET NOT NULL
Specifies that the column cannot contain null values. All values for this column in existing rows of the table must be not null. If the specified column has a default value and a new default value is not specified, the existing default value must not be NULL. SET NOT NULL is not allowed if the column is identified in the foreign key of a referential constraint with a DELETE rule of SET NULL and no other nullable columns exist in the foreign key.

SET NOT HIDDEN or IMPLICITLY HIDDEN
Specifies the hidden attribute for the column.

NOT HIDDEN
Indicates the column is included in implicit references to the table in SQL statements.

IMPLICITLY HIDDEN
Indicates the column is not visible in SQL statements unless it is referred to explicitly by name. For example, SELECT * does not include any hidden columns in the result. A table must contain at least one column that is not IMPLICITLY HIDDEN.

SET FIELDPROC
Designates an *external-program-name* as the field procedure exit routine for the column. It must be an ILE program that does not contain SQL. It cannot be a service program.

The field procedure encodes and decodes column values. Before a value is inserted in the column, it is passed to the field procedure for encoding. Before a value from the column is used, it is passed to the field procedure for decoding.

The field procedure is also invoked during the processing of the ALTER TABLE statement. When so invoked, the procedure provides DB2 with the column's field description. The field description defines the data characteristics of the encoded values. By contrast, the information supplied for the column defines the data characteristics of the decoded values.

constant
Specifies a parameter that is passed to the field procedure when it is invoked. A parameter list is optional.

A field procedure cannot be defined for a column that is a ROWID or DATALINK or a distinct type based on a ROWID or DATALINK. The column must not be an identity column or a row change timestamp column. The column must not have a default value of CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP, or USER. The nullability attribute of the encoded and decoded form of the field must match. The column cannot be referenced in a check condition, unless it is referenced in a NULL predicate. If it is part of a foreign key, the corresponding parent key column must use the same field procedure. See SQL Programming topic collection for an example of a field procedure.

DROP DEFAULT
Drops the current default for the column. The specified column must have a default value and must not have NOT NULL as the null attribute. The new default value is the null value.

DROP NOT NULL
Drops the NOT NULL attribute of the column, allowing the column to have the null value. If a default value is not specified or does not already exist, the new default value is the null value. DROP NOT NULL is not allowed if the column is specified in the primary key of the table or is an identity column, row change timestamp column, or ROWID.

DROP IDENTITY
Drops the identity attributes of the column, making the column a simple numeric data type column. DROP IDENTITY is not allowed if the column is not an identity column.

DROP ROW CHANGE TIMESTAMP
Drops the row change timestamp attribute of the column, making the column a simple timestamp column. DROP ROW CHANGE TIMESTAMP is not allowed if the column is not a row change timestamp column.

DROP FIELDPROC
Drops the field procedure for the column. DROP FIELDPROC is not allowed if the column does not have a field procedure defined.

identity-alteration
Alters the identity attributes of the column. The column must exist in the specified table and must already be defined with the IDENTITY attribute. For a description of the attributes, see AS IDENTITY.

RESTART
Specifies the next value for an identity column. If WITH *numeric-constant* is not specified, the sequence is restarted at the value specified implicitly or explicitly as the starting value when the identity column was originally created. RESTART does not change the original START WITH value.

WITH *numeric-constant*
Specifies that *numeric-constant* will be used as the next value for the column. The *numeric-constant* must be an exact numeric constant that can be any positive or negative value that could be assigned to this column, without nonzero digits existing to the right of the decimal point.

DROP COLUMN

Drops the identified column from the table.

column-name
Identifies the column to be dropped. The column name must not be qualified. The name must identify a column of the specified table. The name must not identify a column that was already added or altered in this ALTER TABLE statement. The name must not identify the only column of a table. The name must not identify a partition key of a partitioned table or a distributed table.

CASCADE
Specifies that any views, indexes, triggers, or constraints that are dependent on the column being dropped are also dropped. [79]

RESTRICT
Specifies that the column cannot be dropped if any views, indexes, triggers, or constraints are dependent on the column. [79]

If all the columns referenced in a constraint are dropped in the same ALTER TABLE statement, RESTRICT does not prevent the drop.

ADD unique-constraint

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not be the same as a constraint name that was previously specified in the ALTER TABLE statement and must not identify a constraint that already exists at the current server. The *constraint-name* must be unique within a schema.

If not specified, a unique constraint name is generated by the database manager.

UNIQUE (*column-name***,...)**
Defines a unique constraint composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB, DATALINK, or XML column. If a sort sequence is specified, the column must not contain a field procedure. The number of identified columns must not exceed 120, and the sum of their lengths must not exceed 32766-*n*, where *n* is the number of columns specified that allow nulls.

The set of identified columns cannot be the same as the set of columns specified in another UNIQUE constraint or PRIMARY KEY on the table. For example, UNIQUE (A,B) is not allowed if UNIQUE (B,A) or PRIMARY KEY (A,B) already exists on the table. Any existing nonnull values in the set of columns must be unique. Multiple null values are allowed.

If a unique index already exists on the identified columns, that index is designated as a unique constraint index. Otherwise, a unique index is created to support the uniqueness of the unique key. The unique index is created as part of the system physical file, not as a separate system logical file.

PRIMARY KEY (*column-name***,...)**
Defines a primary key composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB, DATALINK, or XML column. If a sort sequence is specified, the column must not contain a field procedure. The number of identified columns must not exceed 120, and the sum of their lengths must not exceed 32766. The table must not already have a primary key.

79. A trigger is dependent on the column if it is referenced in the UPDATE OF column list or anywhere in the triggered action.

The identified columns cannot be the same as the columns specified in another UNIQUE constraint on the table. For example, PRIMARY KEY (A,B) is not allowed if UNIQUE (B,A) already exists on the table. Any existing values in the set of columns must be unique.

When a primary key is added, a CHECK constraint is implicitly added to enforce the rule that the NULL value is not allowed in any of the columns that make up the primary key.

If a unique index already exists on the identified columns, that index is designated as a primary index. Otherwise, a primary index is created to support the uniqueness of the primary key. The unique index is created as part of the system physical file, not a separate system logical file.

ADD referential-constraint

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that already exists at the current server.

If not specified, a unique constraint name is generated by the database manager.

FOREIGN KEY
Defines a referential constraint. FOREIGN KEY is not allowed if the table is a declared global temporary table or a distributed table.

Let T1 denote the table being altered.

(*column-name***,...)**
The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T1. The same column must not be identified more than once. The column must not be a LOB, DATALINK, or XML column and must not be a row change timestamp column. If a sort sequence is specified, the column must not contain a field procedure. The number of the identified columns must not exceed 120, and the sum of their lengths must not exceed 32766-*n*, where *n* is the number of columns specified that allows nulls.

REFERENCES *table-name*
The *table-name* specified in a REFERENCES clause must identify a base table that exists at the current server, but it must not identify a catalog table, a declared temporary table, or a distributed table. If the parent is a partitioned table, the unique index that enforces the parent unique constraint must be non-partitioned. This table is referred to as the parent table in the constraint relationship.

A referential constraint is a *duplicate* if its foreign key, parent key, and parent table are the same as the foreign key, parent key, and parent table of an existing referential constraint on the table. Duplicate referential constraints are allowed, but not recommended.

Let T2 denote the identified parent table.

(*column-name***,...)**
The parent key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The same column must not be identified more than once. The column must not be a LOB, DATALINK, or XML column and must not be a row change timestamp column. If a sort sequence is specified, the

column must not contain a field procedure. The number of identified columns must not exceed 120, and the sum of their lengths must not exceed 32766-*n*, where *n* is the number of columns specified that allow nulls.

The list of column names must be identical to the list of column names in the primary key of T2 or a UNIQUE constraint that exists on T2. The names may be specified in any order. For example, if (A,B) is specified, a unique constraint defined as UNIQUE (B,A) would satisfy the requirement. If a column name list is not specified then T2 must have a primary key. Omission of the column name list is an implicit specification of the columns of that primary key.

The specified foreign key must have the same number of columns as the parent key of T2. The description of the *n*th column of the foreign key and the *n*th column of the parent key must have identical data types, lengths, CCSIDs, and FIELDPROCs.

Unless the table is empty, the values of the foreign key must be validated before the table can be used. Values of the foreign key are validated during the execution of the ALTER TABLE statement. Therefore, every nonnull value of the foreign key must match some value of the parent key of T2.

The referential constraint specified by the FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent.

ON DELETE
Specifies what action is to take place on the dependent tables when a row of the parent table is deleted. There are five possible actions:

- NO ACTION (default)
- RESTRICT
- CASCADE
- SET NULL
- SET DEFAULT

SET NULL must not be specified unless some column of the foreign key allows null values. SET NULL and SET DEFAULT must not be specified if T1 has an update trigger.

CASCADE must not be specified if T1 has a delete trigger.

CASCADE must not be specified if T1 contains a DataLink column with FILE LINK CONTROL.

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.
- If SET DEFAULT is specified, each column of the foreign key of each dependent of *p* in T1 is set to its default value.

ON UPDATE
Specifies what action is to take place on the dependent tables when a row of the parent table is updated.

The update rule applies when a row of T2 is the object of an UPDATE or propagated update operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are updated.

ADD check-constraint

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that already exists at the current server. The *constraint-name* must be unique within a schema.

If not specified, a unique constraint name is generated by the database manager.

CHECK(*check-condition***)**
Defines a check constraint. The *check-condition* must be true or unknown for every row of the table.

The *check-condition* is a *search-condition*, except:

- It can only refer to columns of the table and the column names must not be qualified.
- The result of any expression in the *check-condition* cannot be a ROWID, XML, or DATALINK with FILE LINK CONTROL data type. It cannot reference any column that has a field procedure, unless the column is referenced in a NULL predicate.
- It must not contain any of the following:
  - Subqueries
  - Aggregate functions
  - Variables
  - Global variables
  - Parameter markers
  - *Sequence-references*
  - Complex expressions that contain LOBs (such as concatenation)
  - OLAP specifications
  - REGEXP_LIKE predicate
  - ROW CHANGE expressions
  - Special registers
  - Any function that is not deterministic
  - User-defined functions other than functions that were implicitly generated with the creation of a distinct type
  - The following built-in scalar functions:

ATAN2	DLURLCOMPLETE [1]	LPAD	REPLACE
CARDINALITY	DLURLPATH	MAX_CARDINALITY	ROUND_TIMESTAMP
CONTAINS	DLURLPATHONLY	MONTHNAME	RPAD
CURDATE	DLURLSCHEME	MONTHS_BETWEEN	SCORE
CURTIME	DLURLSERVER	NEXT_DAY	SOUNDEX
DATAPARTITIONNAME	DLVALUE	NOW	TIMESTAMP_FORMAT

DATAPARTITIONNUM	ENCRYPT_AES	OVERLAY	TIMESTAMPDIFF
DAYNAME	ENCRYPT_RC2	RAISE_ERROR	TRUNC_TIMESTAMP
DBPARTITIONNAME	ENCRYPT_TDES	RAND	VARCHAR_FORMAT
DECRYPT_BINARY	GENERATE_UNIQUE	REGEXP_COUNT	WEEK_ISO
DECRYPT_BIT	GETHINT	REGEXP_INSTR	XMLPARSE
DECRYPT_CHAR	IDENTITY_VAL_LOCAL	REGEXP_REPLACE	XMLVALIDATE
DECRYPT_DB	INSERT	REGEXP_SUBSTR	XSLTRANSFORM
DIFFERENCE	LOCATE_IN_STRING	REPEAT	

[1] For DataLinks with an attribute of FILE LINK CONTROL and READ PERMISSION DB.

For more information about search-condition, see "Search conditions" on page 225.

DROP

PRIMARY KEY
Drops the definition of the primary key and all referential constraints in which the primary key is a parent key. The table must have a primary key.

UNIQUE *constraint-name*
Drops the unique constraint *constraint-name* and all referential constraints dependent on this unique constraint. The *constraint-name* must identify a unique constraint on the table. DROP UNIQUE will not drop a PRIMARY KEY unique constraint.

FOREIGN KEY *constraint-name*
Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint in which the table is a dependent.

CHECK *constraint-name*
Drops the check constraint *constraint-name*. The *constraint-name* must identify a check constraint on the table.

CONSTRAINT *constraint-name*
Drops the constraint *constraint-name*. The *constraint-name* must identify a unique, referential, or check constraint on the table. If the constraint is a PRIMARY KEY or UNIQUE constraint, all referential constraints in which the primary key or unique key is a parent are also dropped.

CASCADE
Specifies for unique constraints that any referential constraints that are dependent on the constraint being dropped are also dropped.

RESTRICT
Specifies for unique constraints that the constraint cannot be dropped if any referential constraints are dependent on the constraint.

ADD partitioning-clause

Changes a non-partitioned table into a partitioned table. If the specified table is a distributed table or already a partitioned table, an error is returned. A DDS-created physical file cannot be partitioned. See "CREATE TABLE" on page 982 for a description of the *partitioning-clause*.

Changing a non-partitioned table that contains data into a partitioned table will require data movement between the data partitions. When using range partitioning, all existing data in the table must be assignable to the specified range partitions.

DROP PARTITIONING

Changes a partitioned table into a non-partitioned table. If the specified table is already non-partitioned, an error is returned.

Changing a partitioned table that contains data into a non-partitioned table will require data movement between the data partitions.

ADD PARTITION *add-partition*

Adds one or more partitions to a partitioned table. The specified table must be a partitioned table. The number of partitions must not exceed 256.

Changing the number of hash partitions in a partitioned table that contains data will require data movement between the data partitions.

partition-name
Names the partition. A *partition-name* must not identify a data partition that already exists in the table.

If the clause is not specified, a unique partition name is generated by the database manager.

boundary-spec
Specifies the boundaries of a range partition. If the specified table is not a range partitioned table, an error is returned. Both a *starting-clause* and an *ending-clause* must be specified. See "CREATE TABLE" on page 982 for a description of the *boundary-spec*.

integer **HASH PARTITIONS**
Specifies the number of hash partitions to be added. If the specified table is not a hash partitioned table, an error is returned.

ALTER PARTITION

Alters the boundaries of a partition of a range partitioned table. If the specified table is not a range partitioned table, an error is returned.

Changing the boundaries of one or more partitions of a table that contains data may require data movement between the data partitions. All existing data in the table must be assignable to the specified range partitions.

partition-name
Specifies the name of the partition to alter. The *partition-name* must identify a data partition that exists in the table.

boundary-spec
Specifies the new boundaries of a range partition. Both a *starting-clause* and an *ending-clause* must be specified. See "CREATE TABLE" on page 982 for a description of the *boundary-spec*.

DROP PARTITION

Drops a partition of a partitioned table. If the specified table is not a partitioned table, an error is returned. If the last remaining partition of a partitioned table is specified, an error is returned.

partition-name
Specifies the name of the partition to drop. The *partition-name* must identify a data partition that exists in the table.

DELETE ROWS
Specifies that any data in the specified partition will be discarded. All data stored in the partition is dropped from the table without processing any delete triggers.

PRESERVE ROWS
Specifies that any data in the specified partition will be preserved by moving it to the remaining partitions without processing any delete or insert triggers. If the specified table is a range partitioned table, PRESERVE ROWS must not be specified. Dropping a hash partition will require data movement between the remaining data partitions.

ADD MATERIALIZED QUERY materialized-query-definition

Changes a base table to a materialized query table. If the specified table is already a materialized query table or if the table is referenced in another materialized query table, an error is returned.

select-statement
Defines the query on which the table is based. The columns of the existing table must meet the following characteristics:

- The number of columns in the table must be the same as the number of result columns in the *select-statement*.
- The column attributes of each column of the table must be compatible to the column attributes of the corresponding result column in the *select-statement*.

The *select-statement* for a materialized query table must not contain a reference to the table being altered, a view over the table being altered, or another materialized query table. For additional details about specifying *select-statement* for a materialized query table, see "CREATE TABLE" on page 982.

refreshable-table-options
Specifies the materialized query table options for altering a base table to a materialized query table.

DATA INITIALLY DEFERRED
Specifies that the data in the table is not validated as part of the ALTER TABLE statement. A REFRESH TABLE statement can be used to make sure the data in the materialized query table is the same as the result of the query in which the table is based.

DATA INITIALLY IMMEDIATE
Specifies that the data is inserted in the table from the result of the query as part of processing the ALTER TABLE statement.

REFRESH DEFERRED
Specifies that the data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time when the REFRESH TABLE statement is processed or when it was last updated.

MAINTAINED BY USER
Specifies that the materialized query table is maintained by the user. The user can use INSERT, DELETE, UPDATE, or REFRESH TABLE statements on the table.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether this materialized query table can be used for query optimization.

ENABLE QUERY OPTIMIZATION
The materialized query table can be used for query optimization.

DISABLE QUERY OPTIMIZATION
The materialized query table will not be used for query optimization. The table can still be queried directly.

ALTER MATERIALIZED QUERY materialized-query-table-alteration

Changes the attributes of a materialized query table. The *table-name* must identify a materialized query table.

select-statement
Defines the query on which the table is based. The columns of the existing table must meet the following characteristics:

- The number of columns in the table must be the same as the number of result columns in the *select-statement*.
- The column attributes of each column of the table must be compatible to the column attributes of the corresponding result column in the *select-statement*.

The *select-statement* for a materialized query table must not contain a reference to the table being altered, a view over the table being altered, or another materialized query table. For additional details about specifying *select-statement* for a materialized query table, see "CREATE TABLE" on page 982.

refreshable-table-options
Specifies the materialized query table options for altering a base table to a materialized query table.

DATA INITIALLY DEFERRED
Specifies that the data in the table is not refreshed or validated as part of the ALTER TABLE statement. A REFRESH TABLE statement can be used to make sure the data in the materialized query table is the same as the result of the query in which the table is based.

DATA INITIALLY IMMEDIATE
Specifies that the data is inserted in the table from the result of the query as part of processing the ALTER TABLE statement.

REFRESH DEFERRED
Specifies that the data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time when the REFRESH TABLE statement is processed or when it was last updated.

MAINTAINED BY USER
Specifies that the materialized query table is maintained by the user. The user can use INSERT, DELETE, UPDATE, or REFRESH TABLE statements on the table.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether this materialized query table can be used for query optimization.

ENABLE QUERY OPTIMIZATION
The materialized query table can be used for query optimization.

DISABLE QUERY OPTIMIZATION
The materialized query table will not be used for query optimization. The table can still be queried directly.

SET *refreshable-table-alteration*
Changes how the table is maintained or whether the table can be used in query optimization.

MAINTAINED BY USER
Specifies that the materialized query table is maintained by the user. The user can use INSERT, DELETE, UPDATE, or REFRESH TABLE statements on the table.

REFRESH DEFERRED
Specifies that the data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time when the REFRESH TABLE statement is processed or when it was last updated.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether this materialized query table can be used for query optimization.

ENABLE QUERY OPTIMIZATION
The materialized query table can be used for query optimization.

DISABLE QUERY OPTIMIZATION
The materialized query table will not be used for query optimization. The table can still be queried directly.

DROP MATERIALIZED QUERY

Changes a materialized query table so that it is no longer a materialized query table. The table specified by *table-name* must be defined as a materialized query table. The definition of columns and data of the name are not changed, but the table can no longer be used for query optimization and is no longer valid for use with the REFRESH TABLE statement.

ACTIVATE NOT LOGGED INITIALLY

Activates the NOT LOGGED INITIALLY attribute of the table for this current unit of work.

Any changes made to the table by INSERT, DELETE, or UPDATE statements in the same unit of work after the table is altered by this statement are not logged (journaled).

At the completion of the current unit of work, the NOT LOGGED INITIALLY attribute is deactivated and all operations that are done on the table in subsequent units of work are logged (journaled).

ACTIVATE NOT LOGGED INITIALLY is not allowed in a transaction if data change operations are pending for *table-name* or cursors are currently open under commit that reference *table-name*.

ACTIVATE NOT LOGGED INITIALLY is ignored if the table has a DATALINK column with FILE LINK CONTROL or if running with isolation level No Commit (NC).

WITH EMPTY TABLE
Causes all data currently in the table to be removed. If the unit of work in which this ALTER statement was issued is rolled back, the table data will NOT be returned to its original state. When this action is requested, no DELETE triggers defined on the affected table are fired.

WITH EMPTY TABLE cannot be specified for a materialized query table or for a parent in a referential constraint.

A DELETE statement without a WHERE clause will typically perform as well or better than ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE and will allow a ROLLBACK to rollback the delete of the rows in the table.

VOLATILE or NOT VOLATILE

Indicates to the optimizer whether the cardinality of table *table-name* can vary significantly at run time. Volatility applies to the number of rows in the table, not to the table itself. The default is NOT VOLATILE.

VOLATILE
Specifies that the cardinality of *table-name* can vary significantly at run time, from empty to large. To access the table, the optimizer will typically use an index, if possible.

NOT VOLATILE
Specifies that the cardinality of *table-name* is not volatile. Access plans that reference this table will be based on the cardinality of the table at the time the access plan is built.

media-preference

Specifies the preferred storage media for the table or partition.

UNIT ANY
No storage media is preferred. Storage for the table or partition will be allocated from any available storage media. If UNIT ANY is specified on the table, any *media-preference* that is specified on a partition is used. If the table or partition is currently on solid state disk storage, it may be moved asynchronously onto other media, if available.

UNIT SSD
Solid state disk storage media is preferred. Storage for the table or partition may be allocated from solid state disk storage media, if available. If UNIT SSD is specified on the table, any *media-preference* specified on a partition is ignored. If the table or partition is not currently on solid state disk storage, it may be moved asynchronously onto solid state disk storage media, if available.

Notes

Column references: A column can only be referenced *once* in an ADD, ALTER, or DROP COLUMN clause in a single ALTER TABLE statement. However, that same column can be referenced multiple times for adding or dropping constraints in the same ALTER TABLE statement.

Order of operations: The order of operations within an ALTER TABLE statement is:

- drop constraints
- drop materialized query table

- drop partition
- drop partitioning
- drop columns for which the RESTRICT option was specified
- alter all other column definitions
  - drop columns for which the CASCADE option was specified
  - alter column drop attributes (for example, DROP DEFAULT)
  - alter column alter attributes
  - alter column add attributes
  - add columns
- alter partition
- add or alter materialized query table
- add partition or add partitioning
- add constraints

Within each of these stages, the order in which the user specifies the clauses is the order in which they are performed, with one exception. If any columns are being dropped, that operation is logically done before any column definitions are added or altered.

QTEMP considerations: Any views or logical files in another job's QTEMP that are dependent on the table being altered will be dropped as a result of an ALTER TABLE statement.

Authority checking: Authority checking is performed only on the table being altered and any object explicitly referenced in the ALTER TABLE statement (such as tables referenced in the fullselect). Other objects may be accessed by the ALTER TABLE statement, but no authority to those objects is required. For example, no authority is required on views that exist on the table being altered, nor on dependent tables that reference the table being altered through a referential constraint.

Backup recommendation: It is strongly recommended that a current backup of the table and dependent views and logical files exist prior to altering a table.

Performance considerations: The following performance considerations apply to an ALTER TABLE statement when adding, altering, or dropping columns from a table:

- The data in the table may be copied.[80]

  Adding and dropping columns require the data to be copied.

  Altering a column usually requires the data to be copied. The data does not need to be copied, however, if the alter only includes the following changes:
  - The length attribute of a VARCHAR column is increasing and the current length attribute is greater than 20.
  - The length attribute of a VARGRAPHIC column is increasing and the current length attribute is greater than 10.
  - The allocated length of a VARCHAR column is changing and the current and new allocated lengths are both less than or equal to 20.

80. In cases where enough storage does not exist to make a complete copy, a special copy that only requires approximately 16-32 megabytes of free storage is performed.

  - The allocated length of a VARGRAPHIC column is changing and the current and new allocated lengths are both less than or equal to 10.
  - The CCSID of a column is changing but no conversion is necessary between the old and new CCSID. For example, if one CCSID is 65535, no data conversion is necessary.
  - The default value is changing, and the length of the default value is not greater than the current allocated length.
  - DROP DEFAULT is specified.
  - DROP NOT NULL is specified, but at least one nullable column will still exist in the table after the alter table is complete.

  Altering a column that has a field procedure defined might require the field procedure to be run two times.
- Indexes may need to be rebuilt.[81]

  An index does not need to be rebuilt when columns are added to a table or when columns are dropped or altered and those columns are not referenced in the index key.

  Altering a column that is used in the key of an index or constraint usually requires the index to be rebuilt. The index does not need to be rebuilt, however, in the following cases:
  - The length attribute of a VARCHAR or VARGRAPHIC key is increasing.
  - The CCSID of a column is changing but no conversion is necessary between the old and new CCSID. For example, if one CCSID is 65535.

Adding a row change timestamp column: When you add a row change timestamp column to an existing table, the initial values are generated during the alter operation.

Considerations for implicitly hidden columns: A column that is defined as implicitly hidden can be explicitly referenced on the ALTER statement. For example, an implicitly hidden column can be altered, can be specified as part of a referential constraint or a check constraint, or a materialized query table definition.

Altering materialized query tables: The isolation level at the time when a base table is first altered to become a materialized query table by the ALTER TABLE statement is the isolation level for the materialized query table.

Altering a table to change it to a materialized query table with query optimization enabled makes the table eligible for use in optimization. Therefore, ensure that the data in the table is accurate. The DATA INITIALLY IMMEDIATE clause can be used to refresh the data when the table is altered.

Syntax alternatives: The following syntax is supported for compatibility to prior releases. The syntax is non-standard and should not be used:

- INLINE LENGTH is a synonym for ALLOCATE.
- If an ADD constraint is the first clause of the ALTER TABLE statement, the ADD keyword is optional, but strongly recommended. Otherwise, it is required.
- *constraint-name* (without the CONSTRAINT keyword) may be specified following the FOREIGN KEY keywords in a *referential-constraint*.
- PART is a synonym for PARTITION.

81. Any indexes that need to be rebuilt are rebuilt asynchronously by database server jobs.

- PARTITION *partition-number* may be specified instead of PARTITION *partition-name*. A *partition-number* must not identify an existing partition of the table or a partition that was previously specified in the ALTER TABLE statement.

  If a *partition-number* is not specified, a unique partition number is generated by the database manager.
- VALUES is a synonym for ENDING AT.
- SET MATERIALIZED QUERY AS DEFINITION ONLY is a synonym for DROP MATERIALIZED QUERY.
- SET SUMMARY AS DEFINITION ONLY is a synonym for DROP MATERIALIZED QUERY
- SET MATERIALIZED QUERY AS (*select-statement*) is a synonym for ADD MATERIALIZED QUERY (*select-statement*)
- SET SUMMARY AS (*select-statement*) is a synonym for ADD MATERIALIZED QUERY (*select-statement*)

Cascaded Effects

Adding a column has no cascaded effects to SQL views, materialized query tables, or most logical files. For example, adding a column to a table does not cause the column to be added to any dependent views, even if those views were created with a SELECT * clause.

Adding a column does cause any SQL triggers to be recreated and include the new column.

Dropping or altering a column may cause several cascaded effects. Table 72 lists the cascaded effects of dropping a column.

Table 72. Cascaded effects of dropping a column

Operation	RESTRICT Effect	CASCADE Effect
Drop of a column referenced by a view	The drop of the column is not allowed.	The view and all views dependent on that view are dropped.
Drop of a column referenced by a non-view logical file	The drop is allowed, and the column is dropped from the logical file if: • The logical file shares a format with the file being altered, and • The dropped column is not used as a key field or in select/omit specifications, and • That format is not used again in the logical file with another based-on file. Otherwise, the drop of the column is not allowed.	The drop is allowed, and the column is dropped from the logical file if: • The logical file shares a format with the file being altered, and • The dropped column is not used as a key field or in select or omit specifications, and • That format is not used again in the logical file with another based-on file. Otherwise, the logical file is dropped.
Drop of a column referenced in the key of an index	The drop of the index is not allowed.	The index is dropped.

82. A column will also be added to a logical file that shares its physical file's format when a column is added to that physical file (unless that format is used again in the logical file with another based-on file).

Table 72. Cascaded effects of dropping a column (continued)

Operation	RESTRICT Effect	CASCADE Effect
Drop of a column referenced in the key of an keyed physical file where the key is not a PRIMARY KEY	The physical file is changed to a non-keyed physical file.	The physical file is changed to a non-keyed physical file.
Drop of a column referenced in a unique constraint	If all the columns referenced in the unique constraint are dropped in the same ALTER COLUMN statement and the unique constraint is not referenced by a referential constraint, the columns and the constraint are dropped. (Hence, the index used to satisfy the constraint is also dropped.) For example, if column A is dropped, and a unique constraint of UNIQUE (A) or PRIMARY KEY (A) exists and no referential constraints reference the unique constraint, the operation is allowed. Otherwise, the drop of the column is not allowed.	The unique constraint is dropped as are any referential constraints that refer to that unique constraint. (Hence, any indexes used by those constraints are also dropped).
Drop of a column referenced in a referential constraint	If all the columns referenced in the referential constraint are dropped at the same time, the columns and the constraint are dropped. (Hence, the index used by the foreign key is also dropped). For example, if column B is dropped and a referential constraint of FOREIGN KEY (A) exists, the operation is allowed. Otherwise, the drop of the column is not allowed.	The referential constraint is dropped. (Hence, the index used by the foreign key is also dropped).
Drop of a column referenced in an SQL trigger	The drop of the column is not allowed.	The SQL trigger is dropped.
Drop of a column referenced in an MQT	The drop of the column is not allowed.	The MQT is dropped.

Table 73 on page 792 lists the cascaded effects of altering a column. (Alter of a column in the following chart means altering a data type, precision, scale, length, or nullability characteristic.)

Table 73. Cascaded effects of altering a column

Operation	Effect
Alter of a column referenced by a view	The alter is allowed. The views that are dependent on the table will be recreated. The new column attributes will be used when recreating the views.
Alter of a column referenced by a non-view logical file	The alter is allowed. The non-view logical files that are dependent on the table will be recreated. If the logical file shares a format with the file being altered, and that format is not used again in the logical file with another based-on file, the new column attributes will be used when recreating the logical file. Otherwise, the new column attributes will not be used when recreating the logical file. Instead, the current logical file attributes are used.
Alter of a column referenced in the key of an index.	The alter is allowed. (Hence, the index will usually be rebuilt.)
Alter of a column referenced in a unique constraint	The alter is allowed. (Hence, the index will usually be rebuilt.) If the unique constraint is referenced by a referential constraint, the attributes of the foreign keys no longer match the attributes of the unique constraint including the field procedure. The constraint will be placed in a defined and check-pending state.
Alter of a column referenced in a referential constraint	The alter is allowed. • If the referential constraint is in the defined but check-pending state, the alter is allowed and an attempt is made to put the constraint in the enabled state. (Hence, the index used to satisfy the unique constraint will usually to be rebuilt.) • If the referential constraint is in the enabled state, the constraint is placed in the defined and check-pending state.
Alter of a column referenced in an SQL trigger	The trigger is recreated.
Alter of a column referenced in an MQT	The MQT is recreated to include the new attributes.

Examples

Example 1: Add a new column named RATING, which is one character long, to the DEPARTMENT table.

```
ALTER TABLE DEPARTMENT
  ADD RATING CHAR
```

Example 2: Add a new column named PICTURE_THUMBNAIL to the EMPLOYEE table. Create PICTURE_THUMBNAIL as a BLOB column with a maximum length of 1K characters.

```
ALTER TABLE EMPLOYEE
  ADD PICTURE_THUMBNAIL  BLOB(1K)
```

Example 3: Assume a new table EQUIPMENT has been created with the following columns:

EQUIP_NO
INT

EQUIP_DESC
VARCHAR(50)

LOCATION
VARCHAR(50)

EQUIP_OWNER
CHAR(3)

Add a referential constraint to the EQUIPMENT table so that the owner (EQUIP_OWNER) must be a department number (DEPTNO) that is present in the DEPARTMENT table. If a department is removed from the DEPARTMENT table, the owner (EQUIP_OWNER) values for all equipment owned by that department should become unassigned (or set to null). Give the constraint the name DEPTQUIP.

```
ALTER TABLE EQUIPMENT
  FOREIGN KEY DEPTQUIP (EQUIP_OWNER)
    REFERENCES DEPARTMENT
    ON DELETE SET NULL
```

Change the default value for the EQUIP_OWNER column to 'ABC'.

```
ALTER TABLE EQUIPMENT
  ALTER COLUMN EQUIP_OWNER
  SET DEFAULT 'ABC'
```

Drop the LOCATION column. Also drop any views, indexes, or constraints that are built on that column.

```
ALTER TABLE EQUIPMENT
  DROP COLUMN LOCATION CASCADE
```

Alter the table so that a new column called SUPPLIER is added, the existing column called LOCATION is dropped, a unique constraint over the new column SUPPLIER is added, and a primary key is built over the existing column EQUIP_NO.

```
ALTER TABLE EQUIPMENT
  ADD COLUMN SUPPLIER INT
  DROP COLUMN LOCATION
  ADD UNIQUE SUPPLIER
  ADD PRIMARY KEY EQUIP_NO
```

Notice that the column EQUIP_DESC is a variable length column. If an allocated length of 25 was specified, the following ALTER TABLE statement would not change that allocated length.

```
ALTER TABLE EQUIPMENT
  ALTER COLUMN EQUIP_DESC
  SET DATA TYPE VARCHAR(60)
```

Example 4: Alter the EMPLOYEE table. Add the check constraint named REVENUE defined so that each employee must make a total of salary and commission greater than $30,000.

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT REVENUE
  CHECK (SALARY + COMM > 30000)
```

Example 5: Alter EMPLOYEE table. Drop the constraint REVENUE which was previously defined.

```
ALTER TABLE EMPLOYEE
  DROP CONSTRAINT REVENUE
```

Example 6: Alter the EMPLOYEE table. Alter the column PHONENO to accept up to 20 characters for a phone number.

```
ALTER TABLE EMPLOYEE
  ALTER COLUMN PHONENO SET DATA TYPE VARCHAR (20)
```

Example 7: Alter the base table TRANSCOUNT to a materialized query table. The result of the *select-statement* must provide a set of columns that match the columns in the existing table (same number of columns and compatible attributes).

```
ALTER TABLE TRANSCOUNT
  ADD MATERIALIZED QUERY
  (SELECT  ACCTID, LOCID, YEAR, COUNT(*) AS CNT
     FROM TRANS
     GROUP BY ACCTID, LOCID, YEAR )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER
```

ASSOCIATE LOCATORS

The ASSOCIATE LOCATORS statement gets the result set locator value for each result set returned by a procedure.

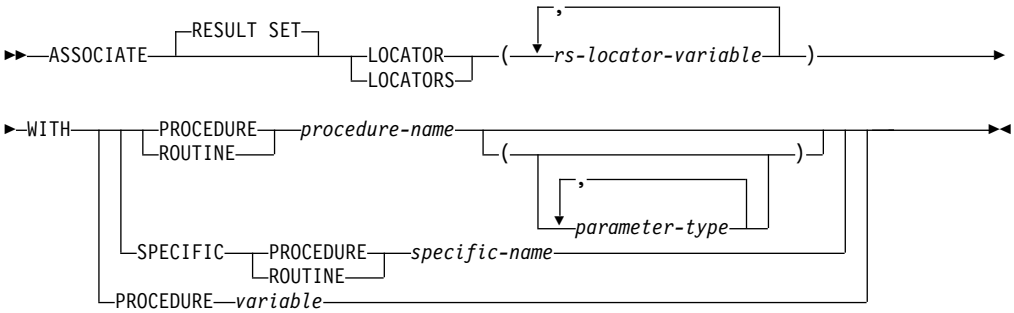
Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared. It cannot be issued interactively. It must not be specified in REXX.

Authorization

None required.

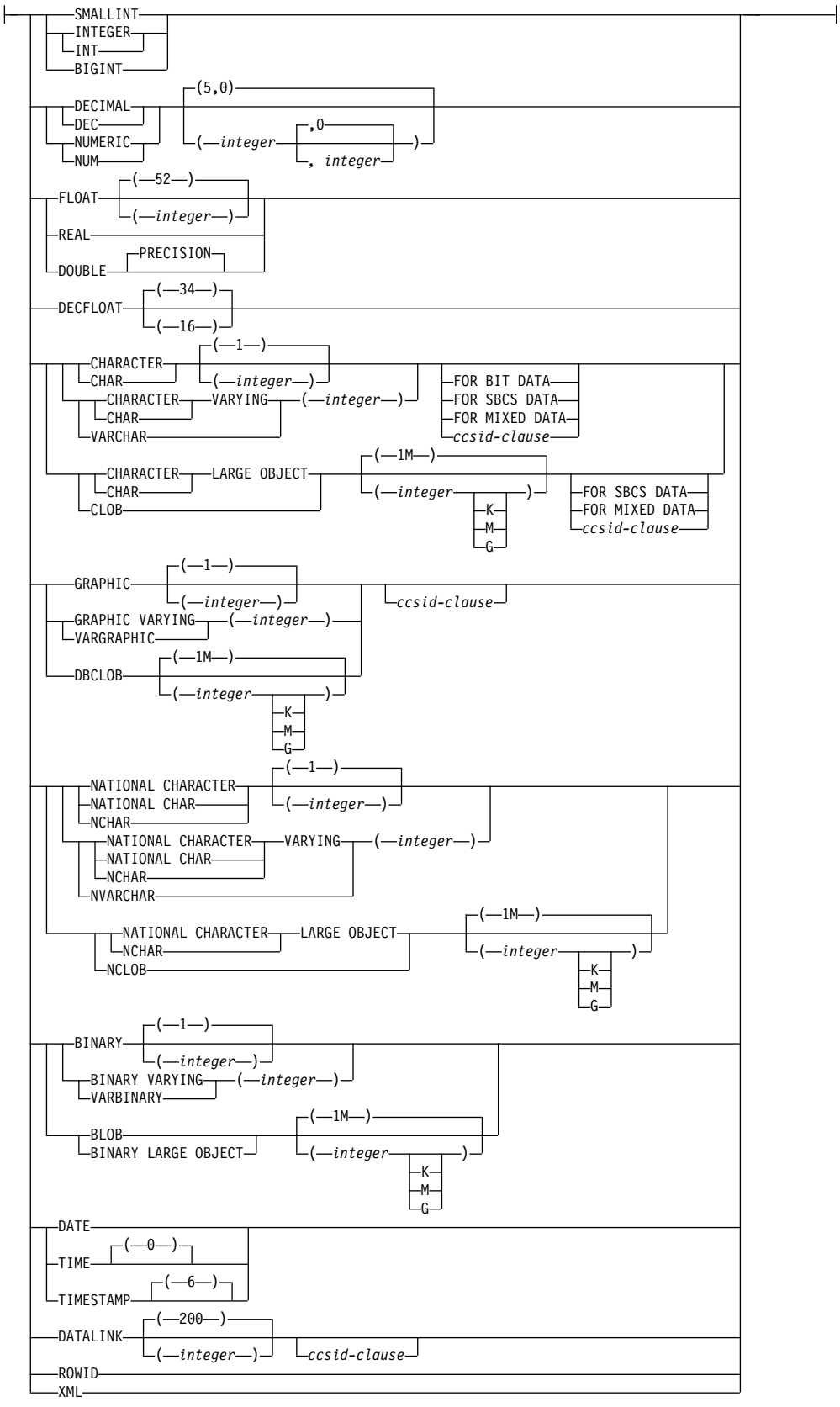
Syntax



parameter-type:

data-type:

built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(integer
,0
, integer
)
FLOAT
(52)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
FOR BIT DATA
FOR SBCS DATA
FOR MIXED DATA
ccsid-clause
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
K
M
G
GRAPHIC
GRAPHIC VARYING
VARGRAPHIC
DBCLOB
NATIONAL CHARACTER
NATIONAL CHAR
NCHAR
NATIONAL CHARACTER VARYING
NATIONAL CHAR VARYING
NCHAR VARYING
NVARCHAR
NATIONAL CHARACTER LARGE OBJECT
NCHAR LARGE OBJECT
NCLOB
BINARY
BINARY VARYING
VARBINARY
BLOB
BINARY LARGE OBJECT
DATE
TIME
(0)
TIMESTAMP
(6)
DATALINK
(200)
ROWID
XML

ccsid-clause:

Description

rs-locator-variable
Identifies a result set locator variable that has been declared according to the rules for declaring result set locator variables.

WITH PROCEDURE *procedure-name* or *variable*
Identifies the procedure that returned one or more result sets. When the ASSOCIATE LOCATORS statement is executed, the procedure name must identify a procedure that the requester has already invoked using the SQL CALL statement.

PROCEDURE or SPECIFIC PROCEDURE
Identifies the procedure. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*
Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be labeled on. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name
Identifies the name of the procedure.

(parameter-type, ...)
Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).

- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

variable
Specifies a variable that contains a procedure or specific name. If *variable* is specified:

- It must be a character-string variable or Unicode graphic-string variable. It cannot be a global variable.
- It must not be followed by an indicator variable.
- The name that is contained within the variable must be left-justified and must be padded on the right with blanks if its length is less than that of the variable.
- The name must be in uppercase unless it is a delimited name.

If only one procedure with this name has been invoked using the CALL statement, the variable is used as a procedure name. If multiple procedures with this name have been invoked, the variable is used as a specific name.

Notes

Assignment of locator values. If a SET RESULT SETS statement was executed in the procedure, the SET RESULT SETS statement identifies the result sets. The locator values are assigned to the items in the descriptor area or the SQLVAR entries in the SQLDA in the order specified on the SET RESULT SETS statement. If a SET RESULT SETS statement was not executed in the procedure, locator values are assigned to the locator variables in the order that the associated cursors are opened at runtime. Locator values are assigned to the locator variables in the same order that they would be placed in the entries in the SQL descriptor area or the SQLDA as a result of a DESCRIBE PROCEDURE statement.

Locator values are not provided for cursors that are closed when control is returned to the invoking application. If a cursor was closed and later re-opened before returning to the invoking application, the most recently executed OPEN CURSOR statement for the cursor is used to determine the order in which the locator values are returned for the procedure result sets. For example, assume procedure P1 opens three cursors A, B, C, closes cursor B, and then issues another OPEN CURSOR statement for cursor B before returning to the invoking application. The locator values assigned for the following ASSOCIATE LOCATORS statement will be in the order A, C, B.

ASSOCIATE RESULT SET LOCATORS (:loc1, :loc2, :loc3) WITH PROCEDURE P1;

More than one locator can be associated with a result set. You can issue multiple ASSOCIATE LOCATORS statements for the same procedure with different result set locator variables to associate multiple locators with each result set.

- If the number of result set locator variables specified in the ASSOCIATE LOCATORS statement is less than the number of result sets returned by the procedure, all locator variables specified in the statement are assigned a value and a warning is issued. For example, assume procedure P1 exists and returns four result sets. Each of the following ASSOCIATE LOCATORS statements returns information on the first result set along with a warning that not enough locators were provided to obtain information about all the result sets.

  CALL P1;

  ASSOCIATE RESULT SET LOCATORS (:loc1) WITH PROCEDURE P1; -> loc1 is assigned a value for first result set, and a warning is returned

  ASSOCIATE RESULT SET LOCATORS (:loc2) WITH PROCEDURE P1; -> loc2 is assigned a value for first result set, and a warning is returned

  ASSOCIATE RESULT SET LOCATORS (:loc3) WITH PROCEDURE P1; -> loc3 is assigned a value for first result set, and a warning is returned

  ASSOCIATE RESULT SET LOCATORS (:loc4) WITH PROCEDURE P1; -> loc4 is assigned a value for first result set, and a warning is returned
- If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is greater than the number of locators returned by the procedure, the extra locator variables are assigned a value of 0.

Multiple calls to the same procedure: When multiple calls to the same procedure are made from the same program, the result sets of earlier invocations are lost unless an ASSOCIATE LOCATOR statement is executed prior to a subsequent CALL to the procedure. The ASSOCIATE LOCATORS statement will refer to the most recent CALL

```
EXEC SQL CALL P1; /* Returns 2 result sets */

EXEC SQL CALL P1; /* Returns 2 result sets, result sets from first invocation are closed */

EXEC SQL ASSOCIATE RESULT SET LOCATORS (:a, :b) WITH PROCEDURE P1; /* Refers to second call */

EXEC SQL CALL P1; /* Returns 2 result sets */

EXEC SQL ASSOCIATE RESULT SET LOCATORS (:c, :d) WITH PROCEDURE P1; /* Refers to third call */

/* The following statements process the result sets from the second call */
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :a;
EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :b;
EXEC SQL FETCH C1 INTO :h1;
EXEC SQL CLOSE C1;
EXEC SQL FETCH C2 INTO :h2;
EXEC SQL CLOSE C2;

/* The following statements process the result sets from the third call */
EXEC SQL ALLOCATE C3 CURSOR FOR RESULT SET :c;
```

```
EXEC SQL ALLOCATE C4 CURSOR FOR RESULT SET :d;
EXEC SQL FETCH C3 INTO :h1;
EXEC SQL CLOSE C3;
EXEC SQL FETCH C4 INTO :h2;
EXEC SQL CLOSE C4;
```

RETURN TO CLIENT procedures: A result set of a RETURN TO CLIENT procedure becomes associated with the highest procedure on the invocation stack. To associate a locator with such a result set, the procedure name of the highest procedure on the invocation stack must be specified.

Example

Allocate result set locators for procedure P1 which returns 3 result sets

```
ASSOCIATE RESULT SET LOCATORS (:loc11, :loc2, :loc3) WITH PROCEDURE P1;
```

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION statement marks the beginning of an SQL declare section. An SQL declare section contains declarations of host variables that are eligible to be used as host variables in SQL statements in a program.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java, RPG, or REXX.

Authorization

None required.

Syntax

Description

The BEGIN DECLARE SECTION statement is used to indicate the beginning of an SQL declare section. It can be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. It cannot be coded in the middle of a host structure declaration. An SQL declare section ends with an END DECLARE SECTION statement, described in "END DECLARE SECTION" on page 1165.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and cannot be nested.

SQL statements should not be included within a declare section, with the exception of the DECLARE VARIABLE and INCLUDE statements.

If SQL declare sections are specified in the program, only the variables declared within the SQL declare sections can be used as host variables. If SQL declare sections are not specified in the program, all variables in the program are eligible for use as host variables.

SQL declare sections should be specified for host languages, other than RPG and REXX, so that the source program conforms to the IBM SQL standard of SQL. SQL declare sections are required for all host variables in C++. The SQL declare section should appear before the first reference to the variable. Host variables are declared without the use of these statements in Java and RPG, and they are not declared at all in REXX.

Variables declared outside an SQL declare section should not have the same name as variables declared within an SQL declare section.

More than one SQL declare section can be specified in the program.

Examples

Example 1: Define the host variables hv_smint (SMALLINT), hv_vchar24 (VARCHAR(24)), and hv_double (DOUBLE) in a C program.

```
EXEC SQL  BEGIN DECLARE SECTION;
  static short                          hv_smint;
  static struct {
           short hv_vchar24_len;
           char  hv_vchar24_value[24];
         }                              hv_vchar24;
  static double                         hv_double;
EXEC SQL  END DECLARE SECTION;
```

Example 2: Define the host variables HV-SMINT (smallint), HV-VCHAR24 (varchar(24)), and HV-DEC72 (dec(7,2)) in a COBOL program.

```
WORKING-STORAGE SECTION.
      EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
01  HV-SMINT              PIC S9(4)       BINARY.
01  HV-VCHAR24.
    49 HV-VCHAR24-LENGTH  PIC S9(4)       BINARY.
    49 HV-VCHAR24-VALUE   PIC X(24).
01  HV-DEC72              PIC S9(5)V9(2)  PACKED-DECIMAL.
      EXEC SQL END DECLARE SECTION  END-EXEC.
```

CALL

The CALL statement calls a procedure.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- If the procedure is an SQL procedure:
  - The EXECUTE privilege on the procedure, and
  - The system authority *EXECUTE on the library containing the SQL procedure
- If the procedure is a Java external procedure:
  - Read authority (*R) to the integrated file system file that contains the Java class.
  - Read and execute authority (*RX) to all directories that must be accessed in order to find the integrated file system file.
- If the procedure is a REXX external procedure:
  - The system authorities *OBJOPR, *READ, and *EXECUTE on the source file associated with the procedure,
  - The system authority *EXECUTE on the library containing the source file, and
  - The system authority *USE to the CL command,
- If the procedure is an external procedure, but not a REXX or Java external procedure:
  - The system authority *EXECUTE on the program or service program associated with the procedure, and
  - The system authority *EXECUTE on the library containing the program or service program associated with the procedure
- Administrative authority

If a global variable is referenced as an IN or INOUT parameter, the privileges held by the authorization ID for the statement must include:

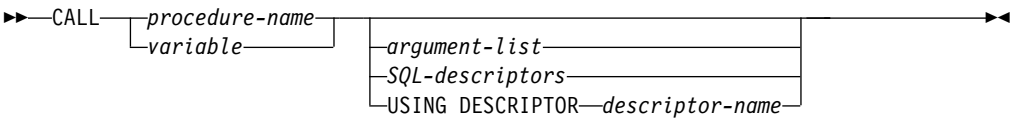
- The READ privilege on the global variable.

If a global variable is referenced as an OUT or INOUT parameter, the privileges held by the authorization ID for the statement must include:

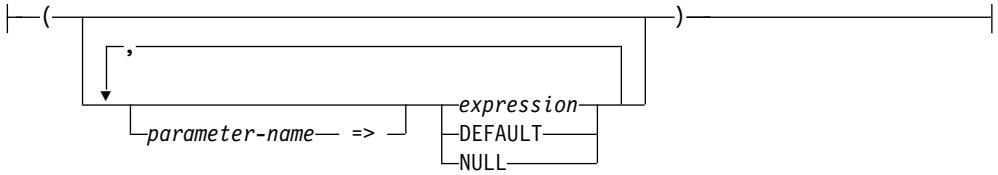
- The WRITE privilege on the global variable.

For information on the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure.

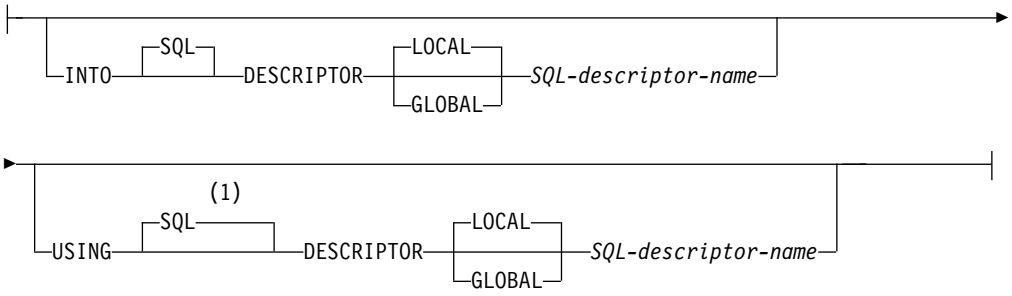
Syntax



argument-list:



SQL-descriptors:



Notes:

1 If an SQL descriptor is specified in the USING clause and the INTO clause is not specified, USING DESCRIPTOR is not allowed and USING SQL DESCRIPTOR must be specified.

Description

procedure-name **or** *variable*

Identifies the procedure to call by the specified *procedure-name* or the procedure name contained in the *variable*. The identified procedure must exist at the current server.

If a *variable* is specified:

- It must be a character-string variable or Unicode graphic-string. It cannot be a global variable.
- It must not be followed by an indicator variable.
- The procedure name that is contained within the variable must be left-justified and must be padded on the right with blanks if its length is less than that of the variable.
- The name of the procedure must be in uppercase unless the procedure name is a delimited name.

If the procedure name is unqualified, it is implicitly qualified based on the path and number of parameters. For more information see "Qualification of unqualified object names" on page 63.

If the procedure name identifies a procedure that was defined by a DECLARE PROCEDURE statement, and the current server is DB2 for i, then:

- The DECLARE PROCEDURE statement determines the name of the external program, language, and calling convention.
- The attributes of the parameters of the procedure are defined by the DECLARE PROCEDURE statement.

If there is no DECLARE PROCEDURE and the procedure name identifies a procedure that was defined by a CREATE PROCEDURE statement, and the current server is DB2 for i, then:

- The CREATE PROCEDURE statement determines the name of the external program, language, and calling convention.
- The attributes of the parameters of the procedure are defined by the CREATE PROCEDURE statement.

Otherwise:

- The current server determines the name of the external program, language, and calling convention.
- If the current server is DB2 for i:
  - The external program name is assumed to be the same as the external procedure name.
  - If the program attribute information associated with the program identifies a recognizable language, then that language is used. Otherwise, the language is assumed to be C.
  - The calling convention is assumed to be GENERAL.
- The application requester assumes all parameters that are variables or parameter markers are INOUT. All parameters that are not variables are assumed to be IN.
- The actual attributes of the parameters are determined by the current server.

  If the current server is a DB2 for i, the attributes of the parameters will be the same as the attributes of the arguments specified on the CALL statement. [83]

argument-list

Identifies a list of values to be passed as parameters to the procedure. The nth value corresponds to the nth parameter in the procedure.

Each parameter defined (using a CREATE PROCEDURE or DECLARE PROCEDURE statement) as OUT must be specified as a variable. A default cannot be specified for an OUT parameter. If a default is used for an INOUT parameter, then the default expression is used to initialize the parameter for input to the procedure. No value is returned for this parameter when the procedure exits.

When a procedure is called, arguments must be specified for all parameters that are not defined to have a default value. When an argument is assigned to a parameter using the named syntax, then all the arguments that follow it must also be assigned using the named syntax.

Any references to date, time, or timestamp special register values in the argument list will use one clock reading for any default expressions and a separate clock reading for any references in the explicit arguments.

83. Note that in the case of decimal constants, leading zeroes are significant when determining the attributes of the argument. Normally, leading zeroes are not significant.

The application requester assumes all parameters that are variables are INOUT parameters except for Java, where it is assumed all parameters that are variables are IN unless the mode is explicitly specified in the variable reference. All parameters that are not variables are assumed to be input parameters. The actual attributes of the parameters are determined by the current server.

parameter-name
Name of the parameter to which the argument value is assigned. The name must match a parameter name defined for the procedure. Named arguments correspond to the same named parameter regardless of the order in which they are specified in the argument list. When an argument is assigned to a parameter by name, all the arguments that follow it must also be assigned by name.

A named argument must be specified only one time (implicitly or explicitly).

Named arguments are not allowed on a call to a procedure that was not defined using a CREATE PROCEDURE statement.

expression
An *expression* of the type described in "Expressions" on page 165, that does not include an aggregate function or column name. If extended indicator variables are enabled, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used for that expression.

DEFAULT
Specifies the default as defined in the CREATE PROCEDURE statement is used as an argument to the procedure. If no default is defined for the parameter, the NULL value is used.

NULL
Specifies a null value as an argument to the procedure.

SQL-descriptors
If SQL descriptors are specified on CALL, a procedure that has IN and INOUT parameters requires an SQL descriptor to be specified in the USING clause; and a procedure that has OUT or INOUT parameters requires an SQL descriptor to be specified in the INTO clause. If all the parameters of the procedure are INOUT parameters, the same descriptor can be used for both clauses. For more information, see Multiple SQL descriptors on CALL.

INTO
Identifies an SQL descriptor which contains valid descriptions of the output variables to be used with the CALL statement. Before the CALL statement is executed, a descriptor must be allocated using the ALLOCATE DESCRIPTOR statement. The COUNT field in the descriptor header must be set to reflect the number of OUT and INOUT parameters for the procedure. The item information, including TYPE, and where applicable, DATETIME_INTERVAL_CODE, LENGTH, DB2_CCSID, PRECISION, and SCALE, must be set for the variables that are used when processing the statement.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation. The information is returned from the descriptor known in this local scope.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the

SQL session. The information is returned from the descriptor known to any program that executes using the same database connection.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "GET DESCRIPTOR" on page 1182 for an explanation of the information that is placed in the SQL descriptor.

USING
Identifies an SQL descriptor which contains valid descriptions of the input variables to be used with the CALL statement. Before the CALL statement is executed, a descriptor must be allocated using the ALLOCATE DESCRIPTOR statement. The COUNT field in the descriptor header must be set to reflect the number of IN and INOUT parameters for the procedure. The item information, including TYPE, and where applicable, DATETIME_INTERVAL_CODE, LENGTH, DB2_CCSID, PRECISION, and SCALE, must be set for the variables that are used when processing the statement. The DATA item and when nulls are used, the INDICATOR item, must be set for the input variables.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "SET DESCRIPTOR" on page 1361 for an explanation of the information in the SQL descriptor.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of variables.

Before the CALL statement is processed, you must set the following fields in the SQLDA. (The rules for REXX are different. For more information, see the Embedded SQL Programming topic collection.)

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to 16 + SQLN*(80), where 80 is the length of an SQLVAR occurrence. If LOBs or distinct types are specified, there must be two SQLVAR entries for each parameter marker and SQLN must be set to two times the number of parameter markers.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameters in the CALL

statement. The *n*th variable described by the SQLDA corresponds to the *n*th parameter marker in the prepared statement. (For a description of an SQLDA, see Appendix D, "SQLDA (SQL descriptor area)," on page 1523.)

Note that RPG/400 does not provide the function for setting pointers. Because the SQLDA uses pointers to locate the appropriate variables, you have to set these pointers outside your RPG/400 application.

The USING DESCRIPTOR clause is not supported for a CALL statement within a Java program.

Notes

Parameter assignments: When the CALL statement is executed, the value of each of its parameters is assigned (using storage assignment rules) to the corresponding parameter of the procedure. A parameter value that is defined to have a default value can be omitted from the argument list when invoking the procedure. Control is passed to the procedure according to the calling conventions of the host language. When execution of the procedure is complete, the value of each parameter of the procedure is assigned (using storage assignment rules for SQL parameters, otherwise using retrieval assignment rules) to the corresponding parameter of the CALL statement defined as OUT or INOUT. For details on the assignment rules, see "Assignments and comparisons" on page 98.

Global variables: A global variable may be specified and will be modified if used as a parameter that is INOUT or OUT.

Procedure signatures: A procedure is identified by its schema, a procedure name, and the number of parameters. This is called a procedure signature, which must be unique within the database. There can be more than one procedure with the same name in a schema, provided that the number of parameters is different for each procedure.

SQL path: A procedure can be invoked by referring to a qualified name (schema and procedure name), followed by an optional list of arguments enclosed by parentheses. A procedure can also be invoked without the schema name, resulting in a choice of possible procedures in different schemas with the same number of parameters. In this case, the SQL path is used to assist in procedure resolution.

Procedure resolution: For details of how procedure resolution is performed, see "Procedure resolution" on page 70.

Cursors and prepared statements in procedures: All cursors opened in the called procedure that are not result set cursors are closed, and all statements prepared in the called procedure are destroyed when the procedure ends. CLOSQLCSR(*ENDACTGRP) can be used to keep cursors open when the procedure ends. For more information, see "SET OPTION" on page 1368.

Result sets from procedures: There are three ways to return result sets from a procedure:

- If a SET RESULT SETS statement is executed in the procedure, the last SET RESULT SETS statement executed in the procedure identifies the result sets. The result sets are returned in the order specified on the SET RESULT SETS statement.
- If a SET RESULT SETS statement is not executed in the procedure,

- If no cursors have specified a WITH RETURN clause, each cursor that the procedure opens and leaves open when it returns identifies a result set. The result sets are returned in the order in which the cursors are opened.
- If any cursors have specified a WITH RETURN clause, each cursor that is defined with the WITH RETURN clause that the procedure opens and leaves open when it returns identifies a result set. The result sets are returned in the order in which the cursors are opened.

When a result set is returned using an open cursor, the rows are returned starting with the current cursor position.

If a cursor result set references an SQL array type, an error might be returned when referencing the result set.

Locks in procedures: All locks that have been acquired in the called procedure are retained until the end of the unit of work.

Errors from procedures: A procedure can return errors (or warnings) using the SQLSTATE like other SQL statements. Applications should be aware of the possible SQLSTATEs that can be expected when invoking a procedure. The possible SQLSTATEs depend on how the procedure is coded. Procedures may also return SQLSTATEs such as those that begin with '38' or '39' if the database manager encounters problems executing the procedure. Applications should therefore be prepared to handle any error SQLSTATE that may result from issuing a CALL statement.

Nesting CALL statements: A program that is executing as a procedure, a user-defined function, or a trigger can issue a CALL statement. When a procedure, user-defined function, or trigger calls a procedure, user-defined function, or trigger, the call is considered to be nested. There is no limit on how many levels procedures and functions can be nested, but triggers can only be nested up to 200 levels deep.

If a procedure returns any query result sets, the result sets are returned to the caller of the procedure. If the SQL CALL statement is nested, the result sets are visible only to the program that is at the previous nesting level. For example, if a client program calls procedure PROCA, which in turn calls procedure PROCB. Only PROCA can access any result sets that PROCB returns; the client program has no access to the query result sets.

When an SQL or an external procedure is called, an attribute is set for SQL data-access that was defined when the procedure was created. The possible values for the attribute are:

```
NONE
CONTAINS
READS
MODIFIES
```

If a second procedure is invoked within the execution of the current procedure, an error is issued if:

- The invoked procedure possibly contains SQL and the invoking procedure does not allow SQL
- The invoked procedure reads SQL data and the invoking procedure does not allow reading SQL data

- The invoked procedure modifies SQL data and the invoking procedure does not allow modifying SQL data

REXX procedures: If the external procedure to be called is a REXX procedure, then the procedure must be declared using the CREATE PROCEDURE or DECLARE PROCEDURE statement.

Variables cannot be used in the CALL statement within a REXX procedure. Instead, the CALL must be the object of a PREPARE and EXECUTE using parameter markers.

Multiple SQL descriptors on CALL: If SQL descriptors are specified on CALL and a procedure has IN or INOUT parameters and OUT or INOUT parameters, two descriptors must be specified. The number of variables that must be allocated in the SQL descriptors depends on how the SQL descriptor attributes are set and the number of each type of parameter.

- If the input SQL descriptor attributes were set using DESCRIBE INPUT and the output SQL descriptor attributes were set using DESCRIBE (OUTPUT), the SQL descriptors will have attributes that match the actual procedure definition at the current server prior to calling the procedure. In this case, the output SQL descriptor will contain one variable for each OUT and INOUT parameter. Likewise, the input SQL descriptor will contain one variable for each IN and INOUT parameter.

  This is the recommended technique for specifying multiple SQL descriptors on a CALL statement.
- Otherwise, the actual procedure definition at the current server is unknown prior to calling the procedure, so each parameter is assumed to be INOUT at the time the procedure is called. This means that both SQL descriptors must be specified, and since each parameter is assumed to be INOUT, they must have the same number of variables and the TYPE, DATETIME_INTERVAL_CODE, LENGTH, DB2_CCSID, PRECISION, and SCALE of each variable in the output SQL descriptor must be exactly the same as the corresponding variable in the input SQL descriptor. Otherwise, an error is returned.

If multiple SQL descriptors are specified, the DATA or INDICATOR items associated with any INOUT parameters in the input SQL descriptor may also be modified when the procedure is called. Thus, a SET DESCRIPTOR may be necessary to reset the DATA and INDICATOR items for such an input SQL descriptor prior to its use in another SQL statement.

Examples

Example 1: Call procedure PGM1 and pass two parameters.

```
CALL PGM1 (:hv1,:hv2)
```

Example 2: In C, invoke a procedure called SALARY_PROCED using the SQLDA named INOUT_SQLDA.

```
struct sqlda *INOUT_SQLDA;

/* Setup code for SQLDA variables goes here */

CALL SALARY_PROC USING DESCRIPTOR :*INOUT_SQLDA;
```

Example 3: A Java procedure is defined in the database using the following statement:

```
CREATE PROCEDURE PARTS_ON_HAND (IN  PARTNUM  INTEGER,
                                OUT COST     DECIMAL(7,2),
                                OUT QUANTITY INTEGER)
                LANGUAGE JAVA PARAMETER STYLE JAVA
                EXTERNAL NAME 'parts!onhand';
```

A Java application calls this procedure on the connection context 'ctx' using the following code fragment:

```
...
int        variable1;
BigDecimal variable2;
Integer    variable3;
...
#sql [ctx] {CALL PARTS_ON_HAND(:IN variable1, :OUT variable2, :OUT variable3)};
...
```

This application code fragment will invoke the Java method *onhand* in class *parts* since the *procedure-name* specified on the CALL statement is found in the database and has the external name 'parts!onhand'.

Example 4: Call procedure PGM2 on relational database BRANCHRDB2 and pass one parameter.

```
CALL BRANCHRDB2.SCHEMA3.PGM2 (:hv1)
```

Example 5: Assume the following procedure exists.

```
CREATE PROCEDURE update_order(
                    IN IN_POID BIGINT,
                    IN IN_CUSTID BIGINT DEFAULT GLOBAL_CUST_ID,
                    IN NEW_STATUS VARCHAR(10) DEFAULT NULL,
                    IN NEW_ORDERDATE DATE DEFAULT NULL,
                    IN NEW_COMMENTS VARCHAR(1000)DEFAULT NULL)...
```

Also assume that the global variable GLOBAL_CUST_ID is set to the value 1002. Call the procedure to change the status of order 5000 for customer 1002 to 'Shipped'. Leave the rest of the order data as it is by allowing the rest of the arguments to default to the null value.

```
CALL update_order (5000, NEW_STATUS => 'Shipped')
```

The customer with ID 1001 has called and indicated that they received their shipment for purchase order 5002 and are satisfied. Update their order.

```
CALL update_order (5002,
          IN_CUSTID => 1001,
          NEW_STATUS => 'Received',
          NEW_COMMENTS => 'Customer satisfied with the order.')
```

CLOSE

The CLOSE statement closes a cursor. If a result table was created when the cursor was opened, that table is destroyed.

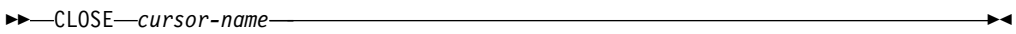
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required. See "DECLARE CURSOR" on page 1079 for the authorization required to use a cursor.

Syntax



Description

cursor-name
Identifies the cursor to be closed. The *cursor-name* must identify a declared cursor as explained in the DECLARE CURSOR statement. When the CLOSE statement is executed, the cursor must be in the open state.

Notes

Implicit cursor close: All cursors in a program are in the closed state when:

- The program is called.
  - If CLOSQLCSR(*ENDPGM) is specified, all cursors are in the closed state each time the program is called.
  - If CLOSQLCSR(*ENDSQL) is specified, all cursors are in the closed state only the first time the program is called as long as one SQL program remains on the call stack.
  - If CLOSQLCSR(*ENDJOB) is specified, all cursors are in the closed state only the first time the program is called in the job.
  - If CLOSQLCSR(*ENDMOD) is specified, all cursors are in the closed state each time the module is initiated.
  - If CLOSQLCSR(*ENDACTGRP) is specified, all cursors are in the closed state the first time the module in the program is initiated within the activation group.
- A program starts a new unit of work by executing a COMMIT or ROLLBACK statement without a HOLD option. Cursors declared with the HOLD option are not closed by a COMMIT statement.

Note: The DB2 for i database manager will open files in order to implement queries. The closing of the files can be separate from the SQL CLOSE statement. For more information, see SQL Programming.

Close cursors for performance: Explicitly closing cursors as soon as possible can improve performance.

Procedure considerations: Special rules apply to cursors within procedures that have not been closed before returning to the calling program. For more information, see "CALL" on page 803.

Example

In a COBOL program, use the cursor C1 to fetch the values from the first four columns of the EMPPROJACT table a row at a time and put them in the following host variables:

- EMP (CHAR(6))
- PRJ (CHAR(6))
- ACT (SMALLINT)
- TIM (DECIMAL(5,2))

Finally, close the cursor.

```
EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.
  77  EMP              PIC X(6).
  77  PRJ              PIC X(6).
  77  ACT              PIC S9(4) BINARY.
  77  TIM              PIC S9(3)V9(2) PACKED-DECIMAL.
EXEC SQL  END DECLARE SECTION  END-EXEC.
.
.
.

EXEC SQL  DECLARE C1 CURSOR FOR
            SELECT EMPNO, PROJNO, ACTNO, EMPTIME
              FROM EMPPROJACT                        END-EXEC.

EXEC SQL  OPEN C1  END-EXEC.

EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM  END-EXEC.

IF SQLSTATE = '02000'
  PERFORM DATA-NOT-FOUND
ELSE
  PERFORM GET-REST-OF-ACTIVITY UNTIL SQLSTATE IS NOT EQUAL TO '00000'.

EXEC SQL  CLOSE C1  END-EXEC.

GET-REST-OF-ACTIVITY
EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM  END-EXEC.
.
.
.
```

COMMENT

The COMMENT statement adds or replaces comments in the catalog descriptions of various database objects.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

To comment on a table, view, alias, column, type, package, sequence, variable, or XSR object, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the table, view, alias, type, package, sequence, variable, or XSR object in the statement,
  - The ALTER privilege on the table, view, alias, type, package, sequence, variable, or XSR object, and
  - The system authority *EXECUTE on the library that contains the table, view, alias, index, type, package, sequence, variable, or XSR object
- Administrative authority

To comment on a constraint or trigger, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the subject table of the constraint or trigger in the statement:
  - The ALTER privilege on the subject table, and
  - The system authority *EXECUTE on the library that contains the subject table
- Administrative authority

To comment on an index, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the index identified in the statement,
  - The system authority *OBJALTER on the index, and
  - The system authority *EXECUTE on the library containing the index.
- Administrative authority

To comment on a function, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSFUNCS and SYSROUTINES catalog view and table:
  - The UPDATE privilege on SYSROUTINES,
  - The system authority *OBJOPR on SYSFUNCS, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To comment on a procedure, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSPROCS and SYSROUTINES catalog view and table:
  - The UPDATE privilege on SYSROUTINES,
  - The system authority *OBJOPR on SYSPROCS, and
  - The system authority *EXECUTE on library QSYS2

- Administrative authority

To comment on a parameter, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSPARMS catalog table:
  - The UPDATE privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To comment on a sequence, the privileges held by the authorization ID of the statement must also include at least one of the following:

- *USE authority to the Change Data Area (CHGDTAARA), CL command
- Administrative authority

To comment on a variable, the privileges held by the authorization ID of the statement must also include at least one of the following:

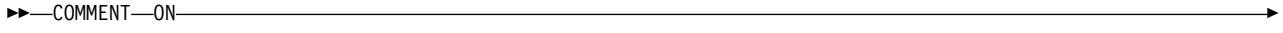
- For the SYSVARIABLES catalog table:
  - The UPDATE privilege on SYSVARIABLES, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To comment on an XSR object, the privileges held by the authorization ID of the statement must also include at least one of the following:

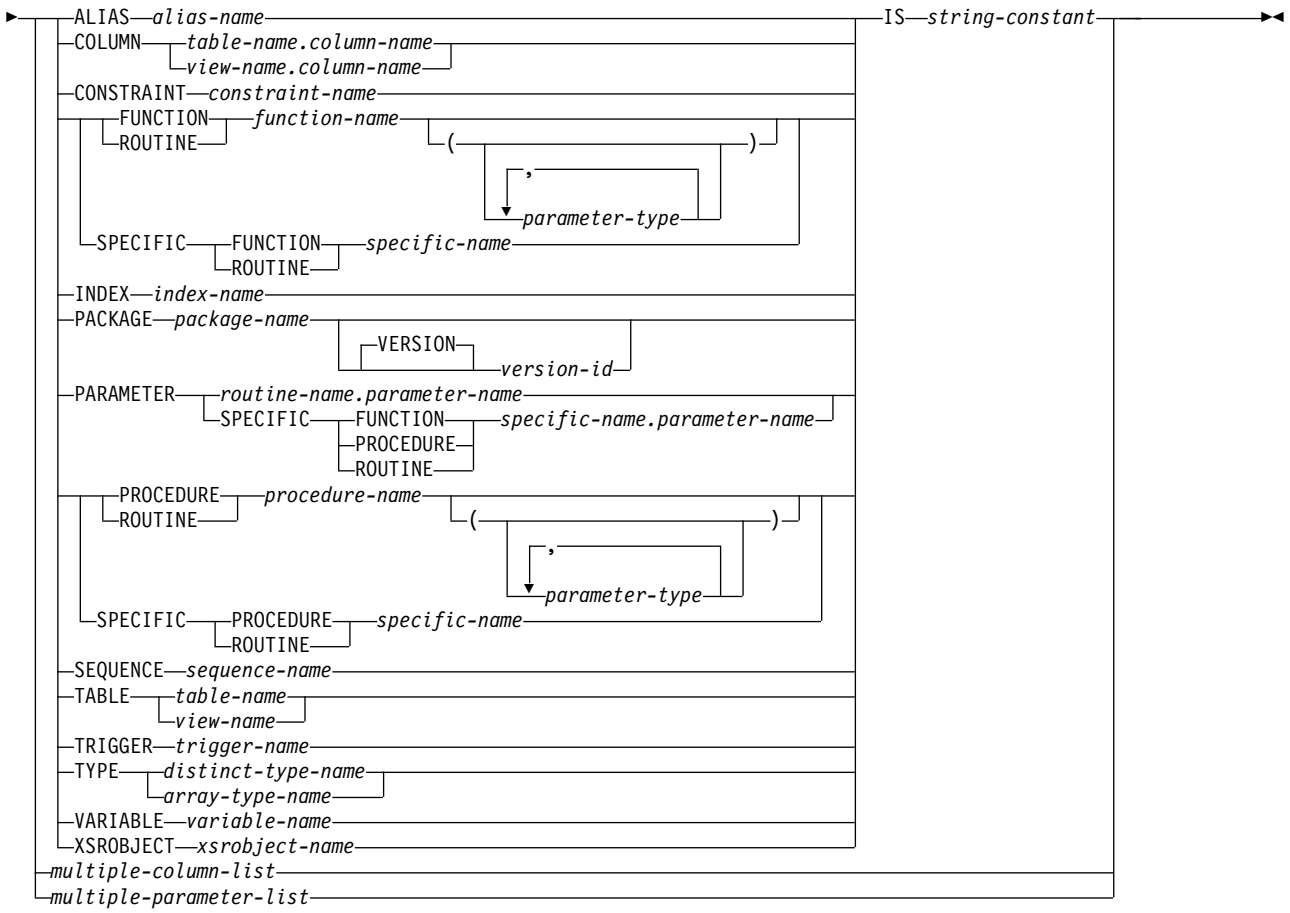
- For the XSROBJECTS catalog table:
  - The UPDATE privilege on XSROBJECTS, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View, Corresponding System Authorities When Checking Privileges to a User-defined Type, Corresponding System Authorities When Checking Privileges to a Sequence, Corresponding System Authorities When Checking Privileges to a Variable , Corresponding System Authorities When Checking Privileges to a Package, and Corresponding System Authorities When Checking Privileges to an XSR object.

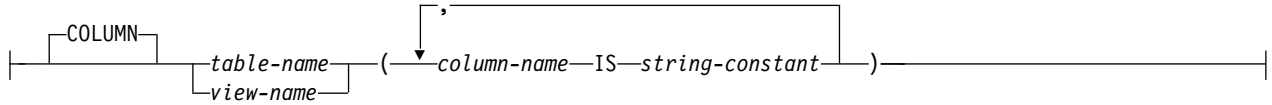
Syntax



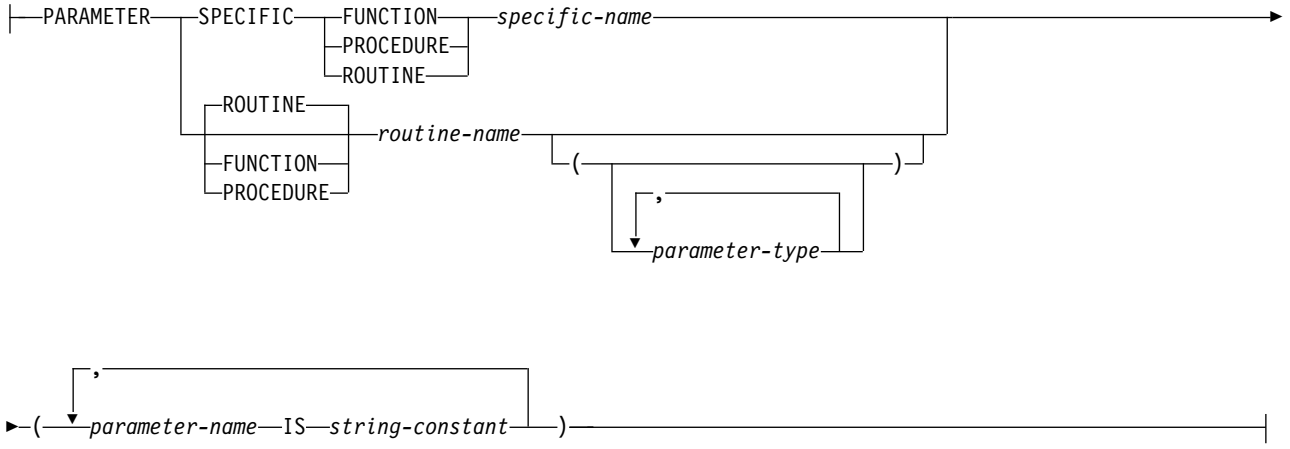
COMMENT



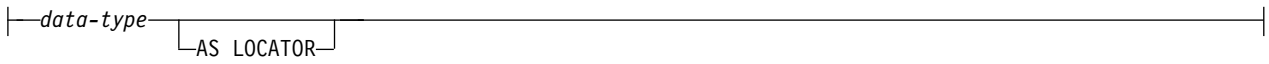
multiple-column-list:



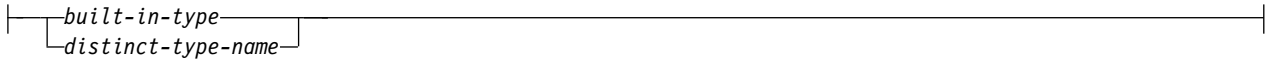
multiple-parameter-list:



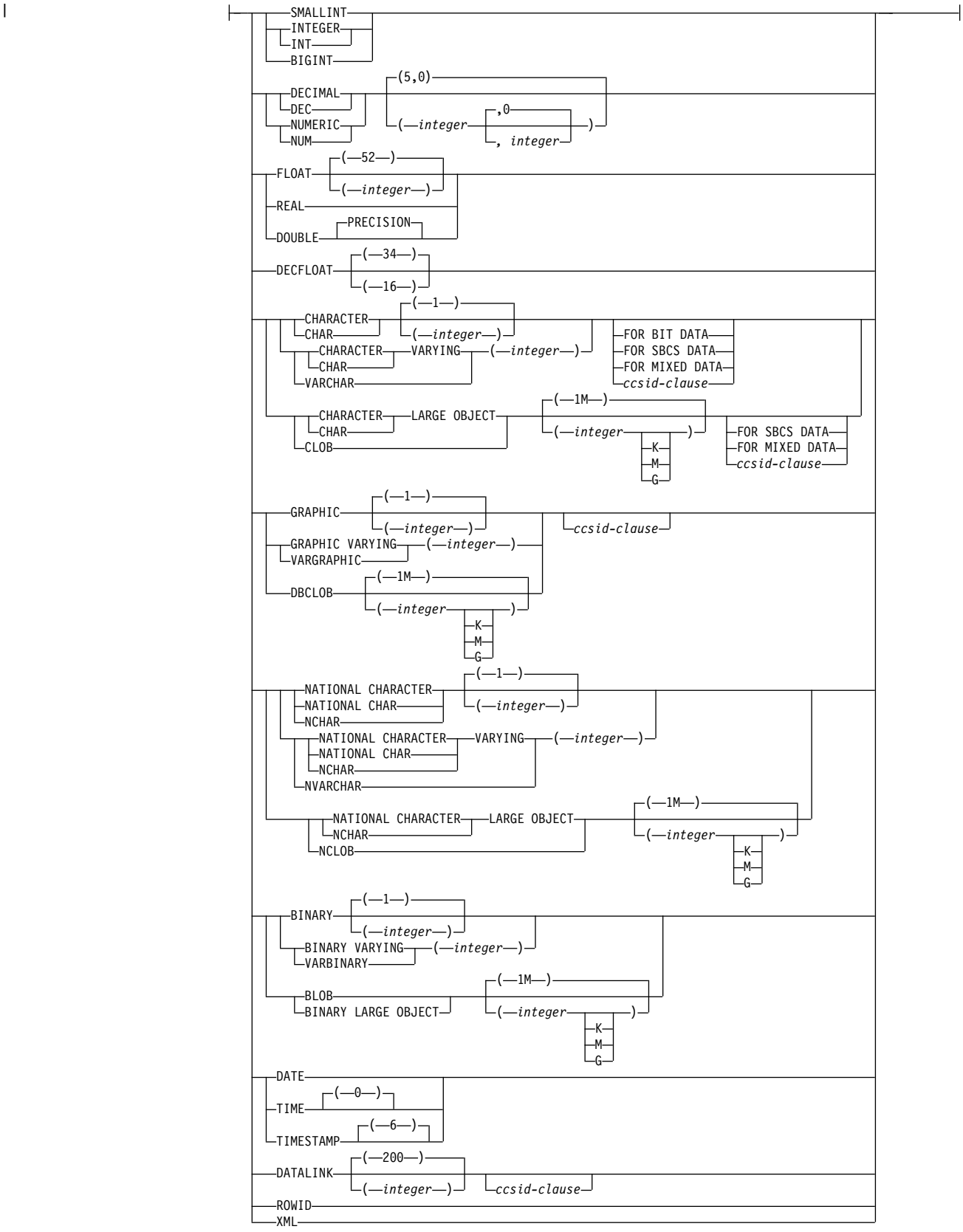
parameter-type:



data-type:



built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(integer
,0
, integer
)
FLOAT
(52)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
FOR BIT DATA
FOR SBCS DATA
FOR MIXED DATA
ccsid-clause
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
K
M
G
GRAPHIC
GRAPHIC VARYING
VARGRAPHIC
DBCLOB
NATIONAL CHARACTER
NATIONAL CHAR
NCHAR
NATIONAL CHARACTER VARYING
NATIONAL CHAR VARYING
NCHAR VARYING
NVARCHAR
NATIONAL CHARACTER LARGE OBJECT
NCHAR LARGE OBJECT
NCLOB
BINARY
BINARY VARYING
VARBINARY
BLOB
BINARY LARGE OBJECT
DATE
TIME
(0)
TIMESTAMP
(6)
DATALINK
(200)
ROWID
XML

ccsid-clause:

Description

ALIAS *alias-name*
Identifies the alias to which the comment applies. The *alias-name* must identify an alias that exists at the current server.

COLUMN
Specifies that a comment will be added to or replaced for a column.

table-name.column-name **or** *view-name.column-name*
Identifies the column to which the comment applies. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a declared temporary table. The *column-name* must identify a column of that table or view.

CONSTRAINT
Specifies that a comment will be added to or replaced for a constraint.

constraint-name
Identifies the constraint to which the comment applies. The *constraint-name* must identify a constraint that exists at the current server.

FUNCTION or SPECIFIC FUNCTION
Identifies the function on which the comment applies. The function must exist at the current server and it must be a user-defined function. The function can be identified by its name, function signature, or specific name.

FUNCTION *function-name*
Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*
Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which to comment. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name
Identifies the name of the function.

(parameter-type, ...)
Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

INDEX *index-name*
Identifies the index to which the comment applies. The *index-name* must identify an index that exists at the current server.

PACKAGE *package-name*
Identifies the package to which the comment applies. The *package-name* must identify a package that exists at the current server. [84]

VERSION *version-id*
version-id is the version identifier that was assigned to the package when it was created. If *version-id* is not specified, a null string is used as the version identifier.

PARAMETER
Specifies that a comment will be added to or replaced for a parameter.

routine-name.parameter-name
Identifies the parameter to which the comment applies. The parameter could be for a procedure or a function. The *routine-name* must identify a procedure or function that exists at the current server, and the *parameter-name* must identify a parameter of that procedure or function.

84. If the identified package has a *version-id*, the comment is limited to 176 bytes.

specific-name.parameter-name
Identifies the parameter to which the comment applies. The parameter could be for a procedure or a function. The *specific-name* must identify a procedure or function that exists at the current server, and the *parameter-name* must identify a parameter of that procedure or function.

PROCEDURE or SPECIFIC PROCEDURE
Identifies the procedure to which the comment applies. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*
Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be commented on. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name
Identifies the name of the procedure.

(parameter-type, ...)
Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

SEQUENCE *sequence-name*
Identifies the sequence to which the comment applies. The *sequence-name* must identify a sequence that exists at the current server.

TABLE *table-name* **or** *view-name*
Identifies the table or view to which the comment applies. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a declared temporary table.

TRIGGER *trigger-name*
Identifies the trigger to which the comment applies. The *trigger-name* must identify a trigger that exists at the current server.

TYPE *distinct-type-name* **or** *array-type-name*
Identifies the distinct type or array type to which the comment applies. The *distinct-type-name* or *array-type-name* must identify a type that exists at the current server.

VARIABLE *variable-name*
Identifies the variable to which the comment applies. The *variable-name* must identify a variable that exists at the current server.

XSROBJECT *xsrobject-name*
Identifies the XSR object to which the comment applies. The *xsrobject-name* must identify an XSR object that exists at the current server.

IS
Introduces the comment that to be added or replaced.

string-constant
Can be any character-string constant of up to 2000 characters (500 for a sequence).

multiple-column-list

To comment on more than one column in a table or view, specify the table or view name and then, in parenthesis, a list of the form:

```
(column-name IS string-constant,
 column-name IS string-constant, ... )
```

The column name must not be qualified, each name must identify a column of the specified table or view, and that table or view must exist at the current server.

multiple-parameter-list

To comment on more than one parameter in a procedure or function, specify the procedure name, function name, or specific name, and then, in parenthesis, a list of the form:

```
(parameter-name IS string-constant,
 parameter-name IS string-constant, ... )
```

The parameter name must not be qualified, each name must identify a parameter of the specified procedure or function, and that procedure or function must exist at the current server.

Notes

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword PROGRAM can be used as a synonym for PACKAGE.
- The keywords DATA TYPE or DISTINCT TYPE can be used as a synonym for TYPE.

Examples

Example 1: Insert a comment for the EMPLOYEE table.

```
COMMENT ON TABLE EMPLOYEE
  IS 'Reflects first quarter 2000 reorganization'
```

Example 2: Insert a comment for the EMP_VIEW1 view.

```
COMMENT ON TABLE EMP_VIEW1
  IS 'View of the EMPLOYEE table without salary information'
```

Example 3: Insert a comment for the EDLEVEL column of the EMPLOYEE table.

```
COMMENT ON COLUMN EMPLOYEE.EDLEVEL
  IS 'Highest grade level passed in school'
```

Example 4: Enter comments on two columns in the DEPARTMENT table.

```
COMMENT ON DEPARTMENT
  (MGRNO IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',
   ADMRDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT')
```

Example 5: Insert a comment for the PAYROLL package.

```
COMMENT ON PACKAGE PAYROLL
  IS 'This package is used for distributed payroll processing.'
```

COMMIT

The COMMIT statement ends a unit of work and commits the database changes that were made by that unit of work.

Invocation

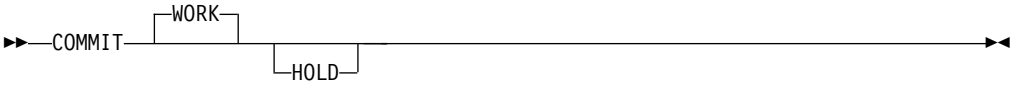
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

COMMIT is not allowed in a trigger if the trigger program and the triggering program run under the same commitment definition. COMMIT is not allowed in a procedure if the procedure is called on a Distributed Unit of Work connection to a remote application server or if the procedure is defined as ATOMIC. COMMIT is not allowed in a function.

Authorization

None required.

Syntax



Description

The COMMIT statement ends the unit of work in which it is executed and starts a new unit of work. It commits all changes made by SQL schema statements (except DROP SCHEMA) and SQL data change statements during the unit of work. For information about SQL schema statements and SQL data change statements see Chapter 6, “Statements,” on page 699.

Connections in the release-pending state are ended.

WORK
COMMIT WORK has the same effect as COMMIT.

HOLD
Specifies a hold on resources. If specified, currently open cursors are not closed whether they are declared with a HOLD option or not. All resources acquired during the unit of work are held. Locks on specific rows and objects implicitly acquired during the unit of work are released.

All implicitly acquired locks are released; except for object level locks required for the cursors that are not closed.

All locators that are not held are released. For more information about held locators, see “HOLD LOCATOR” on page 1248.

Notes

Recommended coding practices: An explicit COMMIT or ROLLBACK statement should be coded at the end of an application process. Either an implicit commit or rollback operation will be performed at the end of an application process

depending on the application environment. Thus, a portable application should explicitly execute a COMMIT or ROLLBACK before execution ends in those environments where explicit COMMIT or ROLLBACK is permitted.

An implicit COMMIT or ROLLBACK may be performed under the following circumstances.

- For the default activation group:
  - An implicit COMMIT is not performed when applications that run in the default activation group end. Interactive SQL, Query Manager, and non-ILE programs are examples of programs that run in the default activation group.
  - In order to commit work, you must issue a COMMIT.
- For non-default activation groups when the scope of the commitment definition is to the activation group:
  - If the activation group ends normally, the commitment definition is implicitly committed.
  - If the activation group ends abnormally, the commitment definition is implicitly rolled back.
- Regardless of the type of activation group, if the scope of the commitment definition is the job, an implicit commit is never performed.

Effect of commit: Commit without HOLD causes the following to occur:

- Connections in the release-pending state are ended.

  For existing connections:
  - all open cursors that were declared with the WITH HOLD clause are preserved and their current position is maintained, although a FETCH statement is required before a Positioned UPDATE or Positioned DELETE statement can be executed.
  - all open cursors that were declared without the WITH HOLD clause including any opened under isolation level NC are closed.
- All LOB locators that are not held are freed. Note that this is true even when the locators are associated with LOB values retrieved via a cursor that has the WITH HOLD property.
- All locks acquired by the LOCK TABLE statement are released. All implicitly acquired locks are released, except for those required for the cursors that were not closed.

Row lock limit: A unit of work can include the processing of up to 4 million rows, including rows retrieved during a SELECT or FETCH statement[85], and rows inserted, deleted, or updated as part of INSERT, DELETE, and UPDATE statements.[86]

Unaffected statements: The commit and rollback operations do not affect the DROP SCHEMA statement, and this statement is not, therefore, allowed in an application program that also specifies COMMIT(*CHG), COMMIT(*CS), COMMIT(*ALL), or COMMIT(*RR).

---

85. This limit also includes:
    - Any rows accessed or changed through files opened under commitment control through high-level language file processing
    - Any rows deleted, updated, or inserted as a result of a trigger or CASCADE, SET NULL, or SET DEFAULT referential integrity delete rule.

86. Unless you specified COMMIT(*CHG) or COMMIT(*CS), in which case these rows are not included in this total.

COMMIT Restrictions: A commit or rollback in a user-defined function in a secondary thread is not allowed.

Commitment definition use: The commitment definition used by SQL is determined as follows:

- If the activation group of the program calling SQL is already using an activation group level commitment definition, then SQL uses that commitment definition.
- If the activation group of the program calling SQL is using the job level commitment definition, then SQL uses the job level commitment definition.
- If the activation group of the program calling SQL is not currently using a commitment definition but the job commitment definition is started, then SQL uses the job commitment definition.
- If the activation group of the program calling SQL is not currently using a commitment definition and the job commitment definition is not started, then SQL implicitly starts a commitment definition. SQL uses the Start Commitment Control (STRCMTCTL) command with:
  - A CMTSCOPE(*ACTGRP) parameter
  - A LCKLVL parameter based on the COMMIT option specified on either the CRTSQLxxx, STRSQL, or RUNSQLSTM commands. In REXX, the LCKLVL parameter is based on the commit option in the SET OPTION statement.

Example

In a C program, transfer a certain amount of commission (COMM) from one employee (EMPNO) to another in the EMPLOYEE table. Subtract the amount from one row and add it to the other. Use the COMMIT statement to ensure that no permanent changes are made to the database until both operations are completed successfully.

```
void main ()
  {

     EXEC SQL  BEGIN DECLARE SECTION;
     decimal(5,2) AMOUNT;
     char FROM_EMPNO[7];
     char TO_EMPNO[7];
     EXEC SQL  END DECLARE SECTION;
     EXEC SQL  INCLUDE SQLCA;
     EXEC SQL  WHENEVER SQLERROR GOTO SQLERR;
     ...
     EXEC SQL  UPDATE EMPLOYEE
               SET COMM = COMM - :AMOUNT
               WHERE EMPNO = :FROM_EMPNO;
     EXEC SQL  UPDATE EMPLOYEE
               SET COMM = COMM + :AMOUNT
               WHERE EMPNO = :TO_EMPNO;
   FINISHED:
     EXEC SQL  COMMIT WORK;
     return;

   SQLERR:
      ...
     EXEC SQL  WHENEVER SQLERROR CONTINUE;  /* continue if error on rollback */
     EXEC SQL  ROLLBACK WORK;
     return;
 }
```

compound (dynamic)

A compound (dynamic) statement groups other statements together in an executable routine. A compound statement allows the declaration of SQL variables, cursors, and condition handlers.

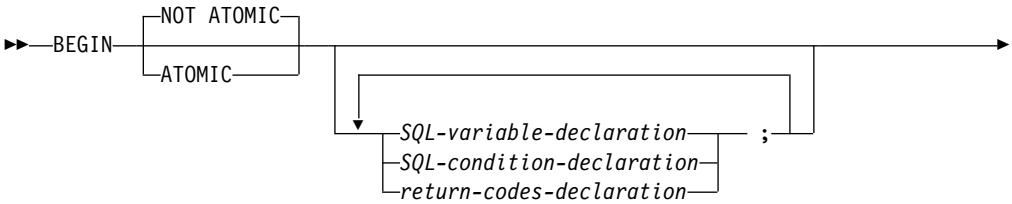
Invocation

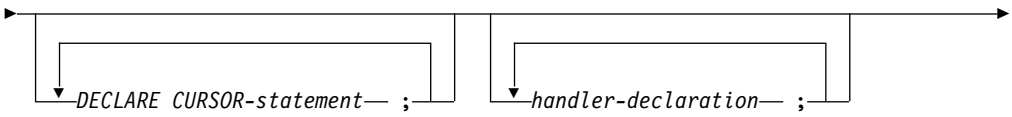
This statement can be issued interactively. It is an executable statement that can be dynamically prepared. It cannot be embedded in an application program.

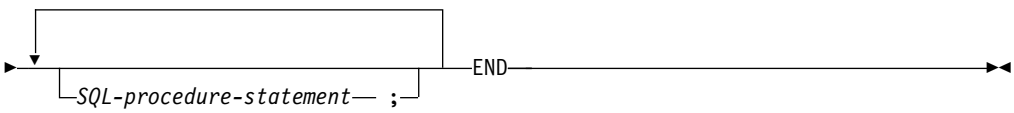
Authorization

The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the compound (dynamic) statement.

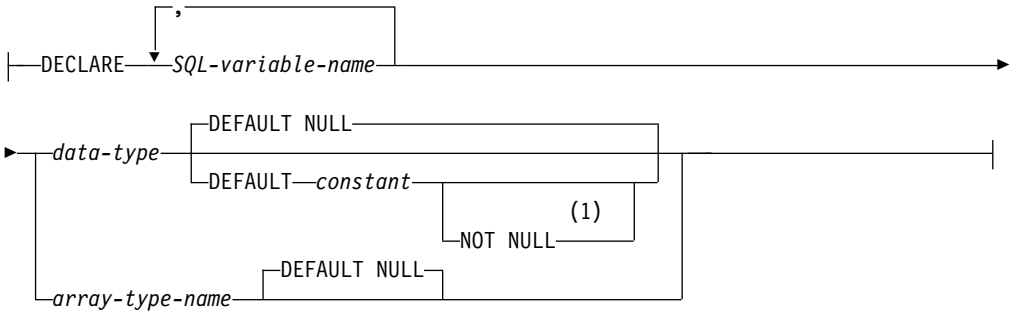
Syntax



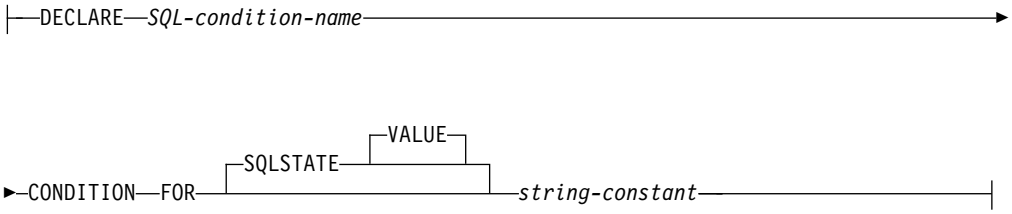




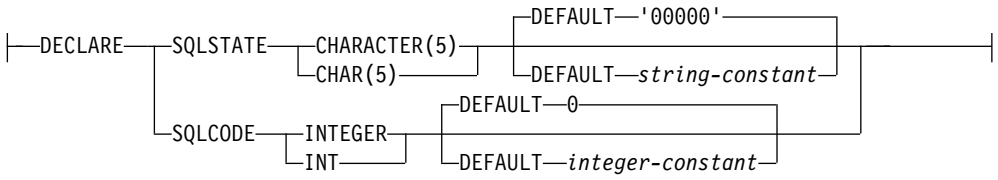
SQL-variable-declaration:



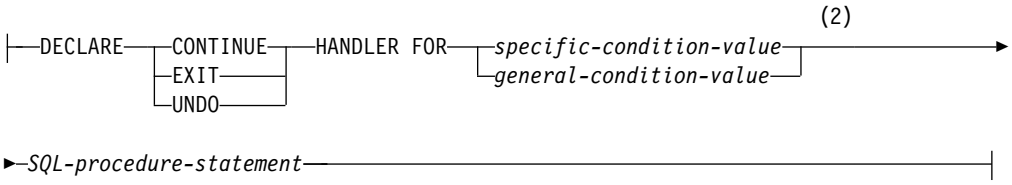
SQL-condition-declaration:



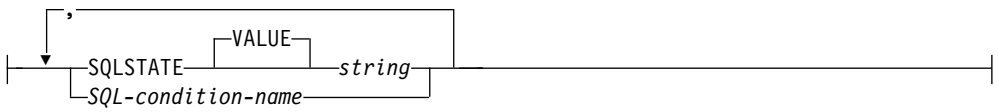
return-codes-declaration:



handler-declaration:



specific-condition-value:



general-condition-value:

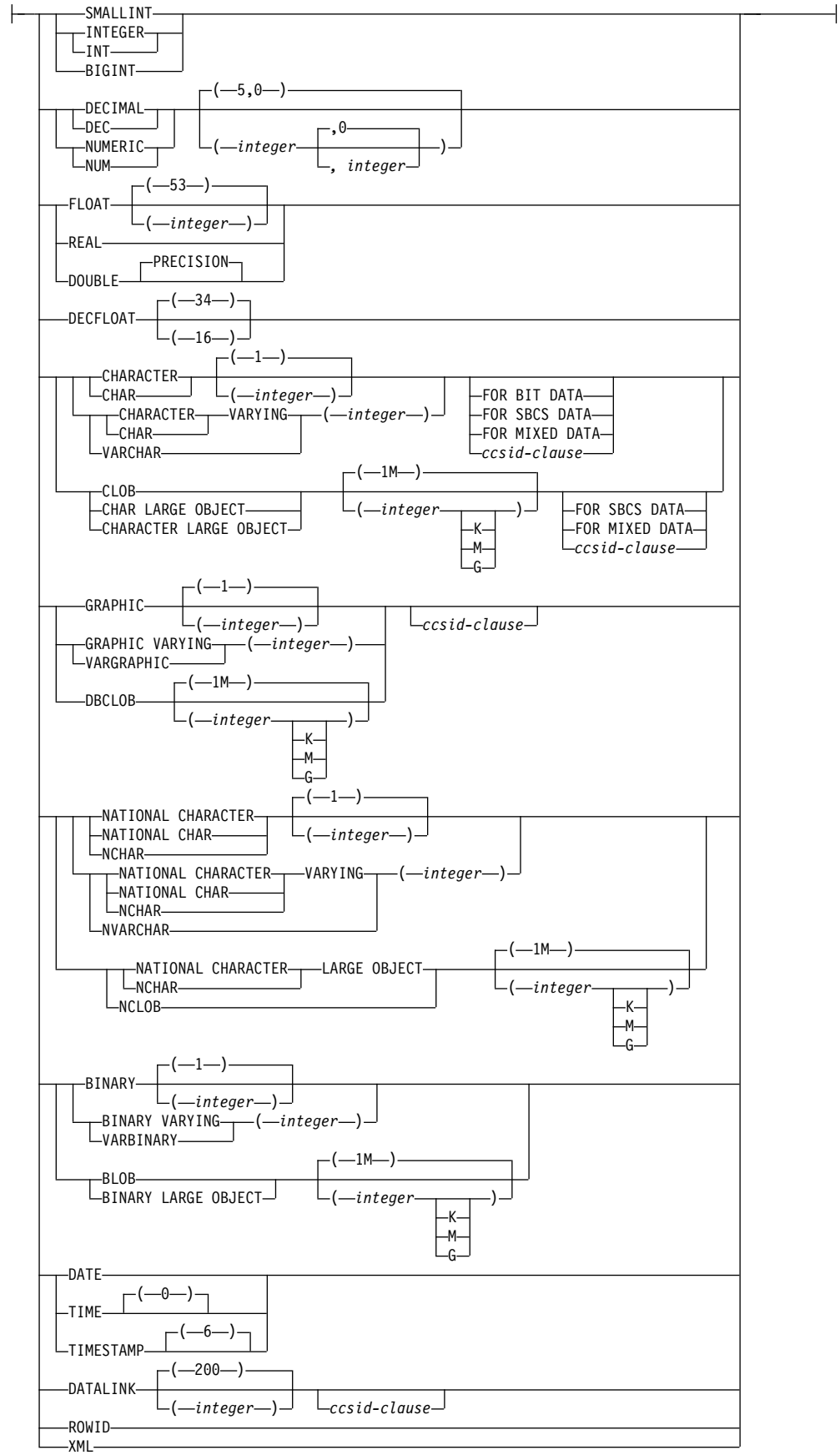
data-type:

Notes:

1 The DEFAULT and NOT NULL clauses can be specified in either order.

2 *specific-condition-value* and *general-condition-value* cannot be specified in the same handler declaration.

built-in-type:



ccsid-clause:

Description

ATOMIC
ATOMIC indicates that an unhandled exception condition within the compound (dynamic) statement causes the compound statement to be rolled back. If ATOMIC is specified, COMMIT or ROLLBACK statements cannot be specified in the compound statement (ROLLBACK TO SAVEPOINT may be specified).

NOT ATOMIC
NOT ATOMIC indicates that an unhandled exception condition within the compound (dynamic) statement does not cause the compound statement to be rolled back.

SQL-variable-declaration
Declares a variable that is local to the compound (dynamic) statement.

SQL-variable-name
Defines the name of a local SQL variable. The database manager converts all undelimited SQL variable names to uppercase. The name must not be the same as another SQL variable within the same compound (dynamic) statement, excluding any declarations in *compound-statements* nested within the compound (dynamic) statement. Do not name SQL variables the same as a column name. See "References to SQL parameters and SQL variables" on page 1429 for how SQL variable names are resolved when there are columns with the same name involved in a statement. Variable names should not begin with 'SQL'.

data-type
Specifies the data type of the variable. See "CREATE TABLE" on page 982 for a description of data type.

If the *data-type* is a graphic string data type, consider specifying CCSID 1200 or 13488 to indicate UTF-16 or UCS-2 data. If a CCSID is not specified, the CCSID of the graphic string variable will be the associated DBCS CCSID for the job.

array-type-name
Specifies that the SQL variable is an array as defined with the CREATE TYPE (Array) statement.

DEFAULT *constant* **or NULL**
Defines the default for the SQL variable. The specified constant must represent a value that could be assigned to the variable in accordance with the rules of assignment as described in "Assignments and comparisons" on page 98. The variable is initialized when the compound (dynamic) statement is invoked. If a default value is not specified, the SQL variable is initialized to NULL. SQL variables of type XML cannot have a default value specified.

NOT NULL
Prevents the SQL variable from containing the NULL value. Omission of NOT NULL implies that the variable can be null. SQL variables of type XML cannot have NOT NULL specified

SQL-condition-declaration
Declares a condition name and corresponding SQLSTATE value.

SQL-condition-name
Specifies the name of the condition. The condition name must be unique within the compound (dynamic) statement, excluding any declarations in *compound-statements* nested within the compound (dynamic) statement.

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE associated with this condition. The string constant must be specified as 5 characters. The SQLSTATE class (the first 2 characters) must not be '00'.

return-codes-declaration
Declares special SQL variables called SQLSTATE and SQLCODE that are set for the first condition in the diagnostics area after executing an SQL statement other than GET DIAGNOSTICS or an empty *compound-statement*.

The SQLSTATE and SQLCODE special variables are only intended to be used as a means of obtaining the SQL return codes that resulted from processing the previous SQL statement other than GET DIAGNOSTICS. If there is any intention to use the SQLSTATE and SQLCODE values, save the values immediately to other SQL variables to avoid having the values replaced by the SQL return codes returned after executing the next SQL statement. If a handler is defined that handles an SQLSTATE, you can use an assignment statement to save that SQLSTATE (or the associated SQLCODE) value in another SQL variable, if the assignment is the first statement in the handler.

Assignment to these variables is not prohibited; however, it is not recommended. Assignment to these special variables is ignored by condition handlers. The SQLSTATE and SQLCODE special variables cannot be set to NULL.

DECLARE CURSOR-statement
Declares a cursor in the compound (dynamic) statement. The cursor name must be unique within the compound (dynamic) statement, excluding any declarations in *compound-statements* nested within the compound (dynamic) statement.

A *cursor-name* can only be referenced within the compound (dynamic) statement in which it is declared, including any *compound-statements* nested within the compound (dynamic) statement.

Use an OPEN statement to open the cursor, and a FETCH statement to read rows using the cursor. Any open cursor is closed at the end of the compound (dynamic) statement.

For more information about declaring a cursor, refer to "DECLARE CURSOR" on page 1079.

handler-declaration
Specifies a *handler*, an *SQL-procedure-statement* to execute when an exception or completion condition occurs in the compound (dynamic) statement.

A condition handler declaration cannot reference the same condition value or SQLSTATE value more than once, and cannot reference an SQLSTATE value and a condition name that represent the same SQLSTATE value. For a list of SQLSTATE values as well as more information, see the SQL messages and codes topic collection.

Furthermore, when two or more condition handlers are declared in a compound (dynamic) statement, no two condition handler declarations may specify the same:

- general condition category or
- specific condition, either as an SQLSTATE value or as a condition name that represents the same value.

A condition handler is active for the set of *SQL-procedure-statements* that follow the *handler-declarations* within the compound (dynamic) statement, including any nested *compound-statements*.

A handler for a condition may exist at several levels of nested compound statements. For example, assume that compound (dynamic) statement *n1* contains another compound statement *n2* which contains another compound statement *n3*. When an exception condition occurs within *n3*, any active handlers within *n3* are first allowed to handle the condition. If no appropriate handler exists in *n3*, then the condition is resignalled to *n2* and the active handlers within *n2* may handle the condition. If no appropriate handler exists in *n2*, then the condition is resignalled to *n1* and the active handlers within *n1* may handle the condition. If no appropriate handler exists in *n1*, the condition is considered unhandled.

There are three types of condition handlers:

CONTINUE
Specifies that after the condition handler is activated and completes successfully, control is returned to the SQL statement following the one that raised the exception. If the error occurs while executing a comparison as in an IF, CASE, FOR, WHILE, or REPEAT, control returns to the statement following the corresponding END IF, END CASE, END FOR, END WHILE, or END REPEAT.

EXIT
Specifies that after the condition handler is activated and completes successfully, control is returned to the end of the compound (dynamic) statement.

UNDO
Specifies that when the condition handler is activated changes made by the compound (dynamic) statement are rolled back. When the handler completes successfully, control is returned to the end of the compound (dynamic) statement. If UNDO is specified, then ATOMIC must be specified.

The conditions under which the handler is activated are:

SQLSTATE *string*
Specifies that the handler is invoked when the specific SQLSTATE occurs. The SQLSTATE class (the first 2 characters) must not be '00'.

SQL-condition-name
Specifies that the handler is invoked when the specific SQLSTATE associated with the condition name occurs. The *SQL-condition-name* must be previously defined in a *SQL-condition-declaration*.

SQLEXCEPTION
Specifies that the handler is invoked when an exception condition occurs. An exception condition is represented by an SQLSTATE value where the first two characters are not '00', '01', or '02'.

SQLWARNING
Specifies that the handler is invoked when a warning condition occurs. A warning condition is represented by an SQLSTATE value where the first two characters are '01'.

NOT FOUND
Specifies that the handler is invoked when a NOT FOUND condition occurs. A NOT FOUND condition is represented by an SQLSTATE value where the first two characters are '02'.

SQL-procedure-statement
An SQL statement or SQL control statement as defined in "SQL-procedure-statement" on page 1435. The SET RESULT SETS and SET SESSION AUTHORIZATION SQL statements are not allowed.

Notes

Compound (dynamic) content: See Chapter 7, "SQL control statements," on page 1427 for more information on the constructs that can be used in a compound (dynamic) statement.

Nesting compound statements: *Compound-statements* can be nested within a compound (dynamic) statement. Nested compound statements can be used to scope variable definitions, condition names, condition handlers, and cursors to a subset of the statements in the compound (dynamic) statement. This can simplify the processing done for each *SQL-procedure-statement*. Support for nested compound statements enables the use of a compound statement within the declaration of a condition handler.

Compound (dynamic) statement execution: When a compound (dynamic) statement is dynamically prepared and executed, the statements within the compound statement are processed as static statements. A temporary program that embeds the statements for the compound statement is created and then executed. The program name is QTEMP.QCMPDnnnnn, where nnnnn is a unique number for the job.

CURRENT PATH and CURRENT SCHEMA: The current schema and current path apply to the prepared and executed compound (dynamic) statement. Any change to these special registers within the compound statement does not affect subsequent resolution of statements within the same compound statement. For example:

```
SET SCHEMA prodlib;
SET stmt =  'BEGIN  SET SCHEMA datalib;  INSERT INTO test_table VALUES(1);  END';
PREPARE compound_stmt FROM stmt;
EXECUTE compound_stmt;
```

When the compound statement `compound_stmt` is prepared and executed, the INSERT will resolve to PRODLIB.TEST_TABLE, not DATALIB.TEST_TABLE.

Most end user SQL script interfaces use dynamic SQL to execute SQL statements. For example, executing the following two statements in IBM i Navigator's Run SQL Scripts is logically equivalent to the above example. The INSERT will resolve to PRODLIB.TEST_TABLE, not DATALIB.TEST_TABLE.

```
SET SCHEMA prodlib;
BEGIN
  SET SCHEMA datalib;
  INSERT INTO test_table VALUES(1);
END;
```

Condition handlers: Condition handlers in a compound (dynamic) statement are similar to WHENEVER statements used in external SQL application programs. A condition handler can be defined to automatically get control when an exception, warning, or not found condition occurs. The body of a condition handler contains code that is executed when the condition handler is activated. A condition handler can be activated as a result of an exception, warning, or not found condition that is returned by the database manager for the processing of an SQL statement, or the activating condition can be the result of a SIGNAL or RESIGNAL statement issued within the routine body.

A condition handler is declared within a compound (dynamic) statement, and it is active for the set of *SQL-procedure-statements* that follow all of the condition handler declarations within the compound (dynamic) statement. To be more specific, the scope of a condition handler declaration H is the list of *SQL-procedure-statements* that follows the condition handler declarations contained within the compound (dynamic) statement in which H appears. This means that the scope of H does not include the statements contained in the body of the condition handler H, implying that a condition handler cannot handle conditions that arise inside its own body. Similarly, for any two condition handlers H1 and H2 declared in the same compound (dynamic) statement, H1 will not handle conditions arising in the body of H2, and H2 will not handle conditions arising in the body of H1.

The declaration of a condition handler specifies the condition that activates it, the type of the condition handler (CONTINUE, EXIT, or UNDO), and the handler action. The type of the condition handler determines where control is returned to after successful completion of the handler action.

Condition handler activation: When a condition other than successful completion occurs in the processing of an *SQL-procedure-statement*, if a condition handler that could handle the condition is within scope, one such condition handler will be activated to process the condition.

In a compound (dynamic) statement with nested *compound-statements*, condition handlers that could handle a specific condition may exist at several levels of the nested compound statements. The condition handler that is activated is a condition handler that is declared innermost to the scope in which the condition was encountered. If more than one condition handler at that nesting level could handle the condition, the condition handler that is activated is the most appropriate handler declared in that compound statement.

The most appropriate handler is a handler that is defined in the compound statement which most closely matches the SQLSTATE of the exception or completion condition.

For example, if the innermost compound statement declares a specific handler for SQLSTATE 22001 as well as a handler for SQLEXCEPTION, the specific handler for SQLSTATE 22001 is the most appropriate handler when an SQLSTATE 22001 is encountered. In this case, the specific handler is activated.

When a condition handler is activated, the condition handler action is executed. If the handler action completes successfully or with an unhandled warning, the diagnostics area is cleared, and the type of the condition handler (CONTINUE, EXIT, or UNDO handler) determines where control is returned. Additionally, the SQLSTATE and SQLCODE SQL variables are cleared when a handler completes successfully or with an unhandled warning.

If the handler action does not complete successfully, and an appropriate handler exists for the condition encountered in the handler action, that condition handler is activated. Otherwise, the condition encountered within the condition handler is unhandled.

Unhandled conditions: If a condition is encountered and an appropriate handler does not exist for that condition, the condition is unhandled.

- If the unhandled condition is an exception, the compound statement containing the failing statement is terminated with an unhandled exception condition.
- If the unhandled condition is a warning or not found condition, processing continues with the next statement. Note that the processing of the next SQL statement will cause information about the unhandled condition in the diagnostics area to be overwritten, and evidence of the unhandled condition will no longer exist.

Considerations for using SIGNAL or RESIGNAL statements with nested compound statements: If an *SQL-procedure-statement* specified in the condition handler is either a SIGNAL or RESIGNAL statement with an exception SQLSTATE, the compound (dynamic) statement terminates with the specified exception. This happens even if this condition handler or another condition handler in the same compound (dynamic) statement specifies CONTINUE, since these condition handlers are not in the scope of this exception. If a *compound-statement* is nested in a compound (dynamic) statement, condition handlers in the compound (dynamic) statement may handle the exception from within the *compound-statement* because those condition handlers are within the scope of the exception.

Null values in SQL variables: If the value of an SQL variable is null and it is used in an SQL statement (such as CONNECT or DESCRIBE) that does not allow an indicator variable, an error is returned.

Effect on open cursors: Upon exit from the compound (dynamic) statement for any reason, all open cursors that are declared in that compound statement are closed.

Considerations for SQLSTATE and SQLCODE SQL variables: The compound (dynamic) statement itself does not affect the SQLSTATE and SQLCODE SQL variables. However, SQL statements contained within the compound statement can affect the SQLSTATE and SQLCODE SQL variables. At the end of the compound statement the SQLSTATE and SQLCODE SQL variables reflect the result of the last SQL statement executed within that compound statement that caused a change to the SQLSTATE and SQLCODE SQL variables. If the SQLSTATE and SQLCODE variables were not changed within the compound (dynamic) statement, they contain the same values as when the compound (dynamic) statement was invoked.

CONNECT (Type 1)

The CONNECT (TYPE 1) statement connects an activation group within an application process to the identified application server using the rules for remote unit of work. This server is then the current server for the activation group. This type of CONNECT statement is used if RDBCNNMTH(*RUW) was specified on the CRTSQLxxx command.

Differences between the two types of statements are described in "CONNECT (Type 1) and CONNECT (Type 2) differences" on page 1511. Refer to "Application-directed distributed unit of work" on page 44 for more information about connection states.

Invocation

This statement can only be embedded within an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

CONNECT is not allowed in a trigger or function.

Authorization

The privileges held by the authorization ID of the statement must include communications-level security. (See the section about security in Distributed Database Programming).

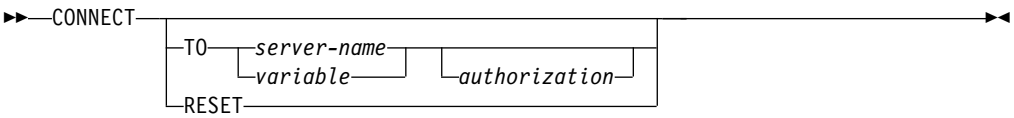
If the application server is DB2 for i, the user profile of the person issuing the statement must also be a valid user profile on the application server system, UNLESS:

- User is specified. In this case, the USER clause must specify a valid user profile on the application server system.
- TCP/IP is used with a server authorization entry for the application server. In this case, the server authorization entry must specify a valid user profile on the application server system.

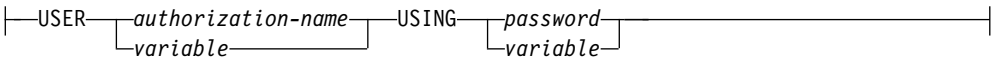
If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



authorization:



Description

TO *server-name* **or** *variable*
Identifies the application server by the specified server name or the server name contained in the variable. It can be a global variable if it is qualified with schema name. If a variable is specified:

- It must be a CHAR or VARCHAR host variable.
- It must not be followed by an indicator variable.
- The server name must be left-justified within the variable and must conform to the rules for forming an ordinary identifier.
- If the length of the server name is less than the length of the variable, it must be padded on the right with blanks.

When the CONNECT statement is executed, the specified server name or the server name contained in the variable must identify an application server described in the local directory and the activation group must be in the connectable state.

If the *server-name* is a local relational database and an *authorization-name* is specified, it must be the user of the job. If the specified *authorization-name* is different than the user of the job, an error occurs and the application is left in the unconnected state.

USER *authorization-name or variable*
Identifies the authorization name that will be used to connect to the application server. It can be a global variable if it is qualified with schema name.

If a *variable* is specified,

- It must be a CHAR or VARCHAR host variable.
- It must not be followed by an indicator variable.
- The authorization name must be left-justified within the variable and must conform to the rules of forming an authorization name.
- If the length of the authorization name is less than the length of the variable, it must be padded on the right with blanks.
- The value of the server name must not contain lowercase characters.

USING *password or variable*
Identifies the password that will be used to connect to the application server.

If password is specified as a literal, it must be a character string. The maximum length is 128 characters. It must be left justified. The literal form of the password is not allowed in static SQL or REXX.

If a *variable* is specified,

- It cannot be a global variable.
- It must be a CHAR or VARCHAR host variable.
- It must not be followed by an indicator variable.
- The password must be left-justified within the variable.
- If the length of the password is less than that of the variable, it must be padded on the right with blanks.

RESET
CONNECT RESET is equivalent to CONNECT TO x where x is the local server name.

CONNECT with no operand
This form of the CONNECT statement returns information about the current server and has no effect on connection states, open cursors, prepared statements, or locks. The connection information is returned in the connection information items in the SQL Diagnostics Area (or the SQLCA).

Notes

Successful connection: If the CONNECT statement is successful:

- All open cursors are closed, all prepared statements are destroyed, and all locks are released from the current connection.
- The activation group is disconnected from all current and dormant connections, if any, and connected to the identified application server.
- The name of the application server is placed in the CURRENT SERVER special register.
- Information about the application server is placed in the *connection-information-items* in the SQL Diagnostics Area.
- Information about the application server is also placed in the SQLERRP and SQLERRD(4) fields of the SQLCA. If the application server is an IBM relational database product, the information in the SQLERRP field has the form *pppvvrrm*, where:
  - *ppp* identifies the product as follows:
    - ARI for DB2 for VM and VSE
    - DSN for DB2 for z/OS
    - QSQ for DB2 for i
    - SQL for all other DB2 products
  - *vv* is a two-digit version identifier such as '09'
  - *rr* is a two-digit release identifier such as '01'
  - *m* is a one-character modification level such as '0'

  For example, if the application server is Version 9 of DB2 for z/OS, the value of SQLERRP is 'DSN09010'.

  The SQLERRD(4) field of the SQLCA contains values indicating whether the application server allows committable updates to be performed. For a CONNECT (Type 1) statement SQLERRD(4) will always contain the value 1. The value 1 indicates that committable updates can be performed, and the connection:
  - Uses an unprotected conversation[87], or
  - Is a connection to an application requester driver program using the *RUW connection method, or
  - Is a local connection using the *RUW connection method.
- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. Refer to Appendix C, “SQLCA (SQL communication area),” on page 1513

87. To reduce the possibility of confusion between network connections and SQL connections, in this book the term 'conversation' will be used to apply to network connections over TCP/IP as well as over APPC, even though it formally applies only to APPC connections.

Unsuccessful connection: If the CONNECT statement is unsuccessful, the DB2_MODULE_DETECTING_ERROR condition information item in the SQL Diagnostics Area (or the SQLERRP field of the SQLCA) is set to the name of the module at the application requester that detected the error. Note that the first three characters of the module name identify the product. For example, if the application requester is DB2 LUW for Windows the first three characters are 'SQL'.

If the CONNECT statement is unsuccessful because the activation group is not in the connectable state, the connection state of the activation group is unchanged.

If the CONNECT statement is unsuccessful for any other reason:

- The activation group remains in a connectable, but unconnected state
- All open cursors are closed, all prepared statements are destroyed, and all locks are released from all current and dormant connections.

An application in a connectable but unconnected state can only execute the CONNECT or SET CONNECTION statements.

Implicit connect:

- When running in the default activation group, the SQL program implicitly connects to a remote relational database when:
  - The activation group is in a connectable state.
  - The first SQL statement in the first SQL program on the program stack is executed.
- When running in a non-default activation group, the SQL program implicitly connects to a remote relational database when the first SQL statement in the first SQL program for that activation group is executed.

Note: It is a good practice for the first SQL statement executed by an activation group to be the CONNECT statement.

When APPC is used for connecting to an RDB, implicit connect always sends the *authorization-name* of the application requester job and does not send passwords. If the *authorization-name* of the application server job is different, or if a password must be sent, an explicit connect statement must be used.

When TCP/IP is used for connecting to an RDB, an implicit connect is not bound by the above restrictions. Use of the ADDSVRAUTE and other -SVRAUTE commands allows one to specify, for a given user under which the implicit (or explicit) CONNECT is done, the remote authorization-name and password to be used in connecting to a given RDB.

In order for the password to be stored with the ADDSVRAUTE or CHGSVRAUTE command, the QRETSVRSEC system value must be set to '1' rather than the default of '0'. When using these commands for DRDA connection, it is very important to realize that the value of the RDB name entered into the SERVER parameter must be in UPPER CASE. For more information, see Example 2 under Type 2 CONNECT.

For more information about implicit connect, refer to the SQL Programming topic collection. Once a connection to a relational database for a user profile is established, the password, if specified, may not be validated again on subsequent connections to the same relational database with the same user profile.

Revalidation of the password depends on if the conversation is still active. See the Distributed Database Programming topic collection for more details.

Connection states: For a description of connection states, see "Remote unit of work connection management" on page 42. Consecutive CONNECT statements can be executed successfully because CONNECT does not remove the activation group from the connectable state.

A CONNECT to either a current or dormant connection in the application group is executed as follows:

- If the connection identified by the server-name was established using a CONNECT (Type 1) statement, then no action is taken. Cursors are not closed, prepared statements are not destroyed, and locks are not released.
- If the connection identified by the server-name was established using a CONNECT (Type 2) statement, then the CONNECT statement is executed like any other CONNECT statement.

CONNECT cannot execute successfully when it is preceded by any SQL statement other than CONNECT, COMMIT, DISCONNECT, SET CONNECTION, RELEASE, or ROLLBACK. To avoid an error, execute a commit or rollback operation before a CONNECT statement is executed.

If any previous current or dormant connections were established using protected conversations, then the CONNECT (Type 1) statement will fail. Either, a CONNECT (Type 2) statement must be used, or the connections using protected conversations must be ended by releasing the connections and successfully committing.

For more information about connecting to a remote relational database and the local directory, see SQL Programming and the Distributed Database Programming.

SET SESSION AUTHORIZATION: If a SET SESSION AUTHORIZATION statement has been executed in the thread, a CONNECT to the local server will fail unless prior to the connect statement, the SYSTEM_USER value is the same as SESSION_USER.

This incudes an implicit connect due to invoking a program that specifies ACTGRP(*NEW).

Examples

Example 1: In a C program, connect to the application server TOROLAB.

```
EXEC SQL  CONNECT TO TOROLAB;
```

Example 2: In a C program, connect to an application server whose name is stored in the variable APP_SERVER (VARCHAR(18)). Following a successful connection, copy the product identifier of the application server to the variable PRODUCT.

```
void main ()
  {
    char product[9] = " ";
    EXEC SQL BEGIN DECLARE SECTION;
    char APP_SERVER[19];
    char username[11];
    char userpass[129];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL INCLUDE SQLCA;
    strcpy(APP_SERVER,"TOROLAB");
```

```
  strcpy(username,"JOE");
  strcpy(userpass,"XYZ1");
  EXEC SQL CONNECT TO :APP_SERVER
           USER :username USING :userpass;
  if (strncmp(SQLSTATE, "00000", 5) )
    { EXEC SQL GET DIAGNOSTICS CONDITION 1
        product = DB2_PRODUCT_ID;  }
  ...
  return;
}
```

CONNECT (Type 2)

The CONNECT (Type 2) statement connects an activation group within an application process to the identified application server using the rules for application directed distributed unit of work. This server is then the current server for the activation group. This type of CONNECT statement is used if RDBCNNMTH(*DUW) was specified on the CRTSQLxxx command.

Differences between the two types of statements are described in "CONNECT (Type 1) and CONNECT (Type 2) differences" on page 1511. Refer to "Application-directed distributed unit of work" on page 44 for more information about connection states.

Invocation

This statement can only be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

CONNECT is not allowed in a trigger or function.

Authorization

The privileges held by the authorization ID of the statement must include communications-level security. (See the section about security in the Distributed Database Programming topic collection.)

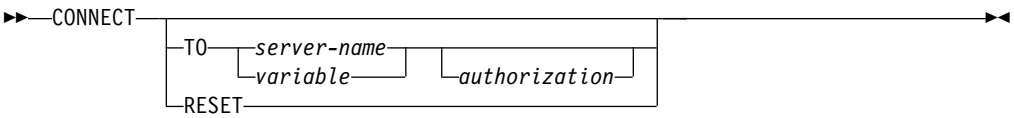
If the application server is DB2 for i, the profile ID of the person issuing the statement must also be a valid user profile on the application server system, UNLESS:

- USER is specified. If USER is specified, the USER clause must specify a valid user profile on the application server system.
- TCP/IP is used with a server authorization entry for the application server. If this is the case, the server authorization entry must specify a valid user profile on the application server system.

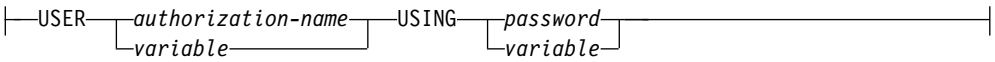
If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



authorization:



Description

TO *server-name* **or** *variable*
Identifies the application server by the specified server name or the server name contained in the variable. It can be a global variable if it is qualified with schema name. If a variable is specified:

- It must be a CHAR or VARCHAR host variable.
- It must not be followed by an indicator variable
- The server name must be left-justified within the variable and must conform to the rules for forming an ordinary identifier
- If the length of the server name is less than the length of the variable, it must be padded on the right with blanks.
- The value of the server name must not contain lowercase characters.

When the CONNECT statement is executed, the specified server name or the server name contained in the variable must identify an application server described in the local directory.

Let S denote the specified server name or the server name contained in the variable. S must not identify an existing connection of the application process.

USER *authorization-name or variable*
Identifies the authorization name that will be used to connect to the application server. It can be a global variable if it is qualified with schema name.

If a *variable* is specified:

- It must be a CHAR or VARCHAR host variable.
- It must not be followed by an indicator variable. The authorization name must be left-justified within the variable and must conform to the rules of forming an authorization name.
- If the length of the authorization name is less than the length of the variable, it must be padded on the right with blanks.

USING *password or variable*
Identifies the password that will be used to connect to the application server.

If password is specified as a literal, it must be a character string. The maximum length is 128 characters. It must be left justified. The literal form of the password is not allowed in static SQL or REXX.

If a *variable* is specified:

- It cannot be a global variable.
- It must be a CHAR or VARCHAR host variable.
- It must not be followed by an indicator variable.
- The password must be left-justified within the variable.
- If the length of the password is less than that of the variable, it must be padded on the right with blanks.

RESET
CONNECT RESET is equivalent to CONNECT TO *x* where *x* is the local server name.

CONNECT with no operand

This form of the CONNECT statement returns information about the current server and has no effect on connection states, open cursors, prepared statements, or locks. The connection information is returned in the connection information items in the SQL Diagnostics Area (or the SQLCA).

In addition, the DB2_CONNECTION_STATUS connection information item in the SQL Diagnostics Area (or the SQLERRD(3) field of the SQLCA) will indicate the status of connection for this unit of work. It will have one of the following values:

- 1 - Committable updates can be performed on the connection for this unit of work.
- 2 - No committable updates can be performed on the connection for this unit of work.

Notes

Successful connection: If the CONNECT statement is successful:

- A connection to application server S is created and placed in the current and held states. The previous connection, if any, is placed in the dormant state.
- S is placed in the CURRENT SERVER special register.
- Information about the application server is placed in the *connection-information-items* in the SQL Diagnostics Area.
- Information about application server S is also placed in the SQLERRP and SQLERRD(4) fields of the SQLCA. If the application server is an IBM relational database product, the information in the SQLERRP field has the form *pppvvrrm*, where:
  - *ppp* identifies the product as follows:
    - ARI for DB2 for VM and VSE
    - DSN for DB2 for z/OS
    - QSQ for DB2 for i
    - SQL for all other DB2 products
  - *vv* is a two-digit version identifier such as '09'
  - *rr* is a two-digit release identifier such as '01'
  - *m* is a one-character modification level such as '0'

  For example, if the application server is Version 9 of DB2 for z/OS, the value of SQLERRP is 'DSN09010'.

  The SQLERRD(4) field of the SQLCA contains values indicating whether application server S allows committable updates to be performed. Following is a list of values and their meanings for the SQLERRD(4) field of the SQLCA on the CONNECT:
  - 1 - committable updates can be performed. Conversation is unprotected. [87]
  - 2 - No committable updates can be performed. Conversation is unprotected.
  - 3 - It is unknown if committable updates can be performed. Conversation is protected.
  - 4 - It is unknown if committable updates can be performed. Conversation is unprotected.
  - 5 - It is unknown if committable updates can be performed. The connection is either a local connection or a connection to an application requester driver program.

- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. Refer to Appendix C, "SQLCA (SQL communication area)," on page 1513.

Unsuccessful connection: If the CONNECT statement is unsuccessful, the connection state of the activation group and the states of its connections are unchanged.

Implicit connect: Implicit connect will always send the *authorization-name* of the application requester job and will not send passwords. If the *authorization-name* of the application server job is different or if a password must be sent, an explicit connect statement must be used.

When TCP/IP is used for connecting to an RDB, an implicit connect is not bound by the above restrictions. Use of the ADDSVRAUTE and other -SVRAUTE commands allows one to specify, for a given user under which the implicit (or explicit) CONNECT is done, the remote authorization-name and password to be used in connecting to a given RDB.

In order for the password to be stored with the ADDSVRAUTE or CHGSVRAUTE command, the QRETSVRSEC system value must be set to '1' rather than the default of '0'. When using these commands for DRDA connection, it is very important to realize that the value of the RDB name entered into the SERVER parameter must be in UPPER CASE. For more information, see Example 2 under Type 2 CONNECT.

For more information about implicit connect, refer to SQL Programming. Once a connection to a relational database for a user profile is established, the password, if specified, may not be validated again on subsequent connections to the same relational database with the same user profile. Revalidation of the password depends on if the conversation is still active. See Distributed Database Programming for more details.

SET SESSION AUTHORIZATION: If a SET SESSION AUTHORIZATION statement has been executed in the thread, a CONNECT to the local server will fail unless prior to the connect statement, the SYSTEM_USER value is the same as SESSION_USER.

This incudes an implicit connect due to invoking a program that specifies ACTGRP(*NEW).

Examples

Example 1: Execute SQL statements at TOROLAB and SVLLAB. The first CONNECT statement creates the TOROLAB connection and the second CONNECT statement places it in the dormant state.

```
EXEC SQL  CONNECT TO TOROLAB;

  (execute statements referencing objects at TOROLAB)

EXEC SQL  CONNECT TO SVLLAB;

  (execute statements referencing objects at SVLLAB)
```

Example 2: Connect to a remote server specifying a userid and password, perform work for the user and then connect as another user to perform further work.

```
EXEC SQL  CONNECT TO SVLLAB USER :AUTHID USING :PASSWORD;

  (execute SQL statements accessing data on the server)

EXEC SQL COMMIT;

  (set AUTHID and PASSWORD to new values)

EXEC SQL  CONNECT TO SVLLAB USER :AUTHID USING :PASSWORD;

  (execute SQL statements accessing data on the server)
```

Example 3: User JOE wants to connect to TOROLAB3 and execute SQL statements under the user ID ANONYMOUS which has a password of SHIBBOLETH. The RDB directory entry for TOROLAB3 specifies *IP for the connection type.

Before running the application, some setup must be done.

This command will be required to allow server security information to be retained in the IBM i operating system, if it has not been previously run:

```
CHGSYSVAL SYSVAL(QRETSVRSEC) VALUE('1')
```

This command adds the required server authorization entry:

```
ADDSVRAUTE USRPRF(JOE) SERVER(TOROLAB3) USRID(ANONYMOUS) +
           PASSWORD(SHIBBOLETH)
```

This statement, run under JOE's user profile, will now make the wanted connection:

```
EXEC SQL CONNECT TO TOROLAB3;
(execute statements referencing objects at TOROLAB3)
```

CREATE ALIAS

The CREATE ALIAS statement defines an alias on a table, partition of a table, view, or member of a database file at the current or remote server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - *USE to the Create DDM File (CRTDDMF) command
- Administrative authority

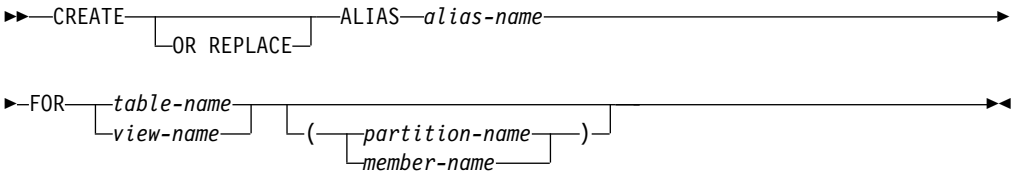
If SQL names are specified and a user profile exists that has the same name as the library into which the alias is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

To replace an existing alias, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority of *OBJMGT on the alias
  - All authorities needed to DROP the alias
- Administrative authority

Syntax



Description

OR REPLACE

Specifies to replace the definition for the alias if one exists at the current server. The existing definition is effectively dropped before the new definition is

replaced in the catalog with the exception that privileges that were granted on the alias are not affected. This option is ignored if a definition for the alias does not exist at the current server.

alias-name
Names the alias. The name, including the implicit or explicit qualifier, must not be the same as an index, table, view, alias or file that already exists at the current server.

If the *alias-name* is qualified, the name can be a two-part or three-part name. The schema name should not be a system schema. If a three-part name is used, the first part must identify a relational database name in the relational database directory.

For more information about connecting to a remote relational database and the local directory, see SQL Programming and the Distributed Database Programming.

If SQL names were specified, the alias will be created in the schema specified by the implicit or explicit qualifier.
If system names were specified, the alias will be created in the schema that is specified by the qualifier. If not qualified, the alias will be created in the same schema as the table or view for which the alias was created. If the table is not qualified and does not exist at the time the alias is created:

- If the value of the CURRENT SCHEMA special register is *LIBL, the alias will be created in the current library (*CURLIB).
- Otherwise, the alias will be created in the current schema.

If the alias name is not a valid system name, DB2 for i will generate a system name. For information about the rules for generating a name, see "Rules for Table Name Generation" on page 1030.

FOR *table-name* **or** *view-name*
Identifies the table or view at the current or remote server for which the alias is to be defined. An alias name cannot be specified (an alias cannot refer to another alias), unless the name refers to an alias on a remote server.

If a three-part name is specified with a non-local relational database name, the *alias-name* must be the same as the *table-name* or *view-name*.

The *table-name* or *view-name* need not identify a table or view that exists at the time the alias is created. If the table or view does not exist when the alias is created, a warning is returned. If the table or view does not exist when the alias is used, an error is returned.

If SQL names were specified and the *table-name* or *view-name* was not qualified, then the qualifier is the implicit qualifier. For more information, see "Naming conventions" on page 54.

If system names were specified and the *table-name* or *view-name* is not qualified and does not exist when the alias is created, the *table-name* or *view-name* is qualified by the library in which the alias is created.

partition-name
Identifies a partition of a partitioned table.

If a partition is specified, the alias cannot be used in SQL schema statements. If a partition is not specified, all partitions in the table are included in the alias.

member-name
Identifies a member of a database file. If a member name is not specified and

the table is not a partitioned table, *FIRST is used. If a member name is not specified and the table is a partitioned table, all members (partitions) are used.

If a member is specified, the alias cannot be used in most SQL schema statements. It can be used in CREATE PROCEDURE, CREATE FUNCTION and in a CREATE TABLE with an *as-result-table* clause.

Notes

The Override Database File (OVRDBF) CL command allows the database manager to process individual members of a database file. Creating an alias over a partition of a table or member of a database file, however, is easier and performs better by eliminating the need to perform the override.

An alias can be defined to reference either the system name or SQL name. However, since system names are generated during create processing, it is recommended that the SQL name be specified.

Alias attributes: An alias is created as a special form of a DDM file. Both an alias and a normal DDM file can be used in SQL, but if the alias or DDM file specifies a non-local relational database name, the specified *table-name* or *view-name* must be the same as the name of the alias.

An alias created over a distributed table is only created on the current server. For more information about distributed tables, see DB2 Multisystem.

Alias ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the alias is created exists, the *owner* of the alias is that user profile.
- Otherwise, the *owner* of the alias is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the alias is the user profile or group user profile of the job executing the statement.

Alias authority: If SQL names are used, aliases are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, aliases are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the alias is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the alias.

REPLACE rules: When an alias is recreated by REPLACE:

- Any existing comment or label is discarded.
- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved.

Packages and three-part aliases: When an application uses three-part name aliases for remote objects and DRDA access, a package for the application program must exist at each location that is specified in the three-part names. A package can be explicitly created using the CRTSQLPKG CL command. If the three-part name alias is referenced and a package does not exist, the database manager will attempt to implicitly create the package.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword SYNONYM can be used as a synonym for ALIAS.

Examples

Example 1: Create an alias named CURRENT_PROJECTS for the PROJECT table.

```
CREATE ALIAS CURRENT_PROJECTS
  FOR PROJECT
```

Example 2: Create an alias named SALES_JANUARY on the JANUARY partition of the SALES table. The sales table has 12 partitions (one for each month of the year).

```
CREATE ALIAS SALES_JANUARY
  FOR SALES(JANUARY)
```

Example 3: Create an alias named REPORTS.SALES for the SALES table in schema REPORTS on relational database USARDB.

```
CREATE ALIAS REPORTS.SALES
  FOR USARDB.REPORTS.SALES
```

CREATE FUNCTION

The CREATE FUNCTION statement defines a user-defined function at the current server.

The following types of functions can be defined:

- **External Scalar**

  The function is written in a programming language such as C or Java and returns a scalar value. The external program is referenced by a function defined at the current server along with various attributes of the function. See "CREATE FUNCTION (External Scalar)" on page 856.
- **External Table**

  The function is written in a programming language such as C or Java and returns a set of rows. The external program is referenced by a function defined at the current server along with various attributes of the function. See "CREATE FUNCTION (External Table)" on page 876.
- **Sourced**

  The function is implemented by invoking another function (built-in, external, sourced, or SQL) that already exists at the current server. A sourced function can return a scalar result, or the result of an aggregate function. See "CREATE FUNCTION (Sourced)" on page 894. The function inherits attributes of the underlying source function.
- **SQL Scalar**

  The function is written exclusively in SQL and returns a scalar value. The function body is defined at the current server along with various attributes of the function. See "CREATE FUNCTION (SQL Scalar)" on page 905.
- **SQL Table**

  The function is written exclusively in SQL and returns a set of rows. The function body is defined at the current server along with various attributes of the function. See "CREATE FUNCTION (SQL Table)" on page 917.

Notes

Choosing the schema and function name: If a qualified function name is specified, the *schema-name* cannot be one of the system schemas (see "Schemas" on page 5). If *function-name* is not qualified, it is implicitly qualified with the default schema name.

The unqualified function name must not be one of the following names reserved for system use even if they are specified as delimited identifiers:

=	<	>	>=
<=	<>	¬=	¬<
¬<	!=	!<	!>
ALL	FALSE	POSITION	XMLAGG
AND	FOR	RID	XMLATTRIBUTES
ANY	FROM	RRN	XMLCOMMENT
ARRAY_AGG	HASHED_VALUE	SELECT	XMLCONCAT
BETWEEN	IN	SIMILAR	XMLDOCUMENT
BOOLEAN	IS	SOME	XMLELEMENT
CASE	LIKE	STRIP	XMLFOREST
CAST	MATCH	SUBSTRING	XMLGROUP
CHECK	NODENAME	TABLE	XMLNAMESPACES

DATAPARTITIONNAME	NODENUMBER	THEN	XMLPARSE
DATAPARTITIONNUM	NOT	TRIM	XMLPI
DBPARTITIONNAME	NULL	TRUE	XMLROW
DBPARTITIONNUM	ONLY	TYPE	XMLSERIALIZE
DISTINCT	OR	UNIQUE	XMLTEXT
EXCEPT	OVERLAPS	UNKNOWN	XMLVALIDATE
EXISTS	PARTITION	WHEN	XSLTRANSFORM
EXTRACT			

Defining the parameters: The input parameters for the function are specified as a list within parenthesis.

The maximum number of parameters allowed in CREATE FUNCTION is 1024 for SQL scalar and SQL table functions and 90 for all other functions.

A function can have no input parameters. In this case, an empty set of parenthesis must be specified, for example:

```
CREATE FUNCTION WOOFER()
```

The data type of the result of the function is specified in the RETURNS clause for the function.

- **Choosing data types for parameters:** When choosing the data types of the input and result parameters for a function, the rules of promotion that can affect the values of the parameters need to be considered. See "Promotion of data types" on page 93. For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, using the following data types is recommended:
  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC

  For portability of functions across platforms that are not DB2 for i, do not use the following data type names, which might have different representations on different platforms:
  - FLOAT. Use DOUBLE or REAL instead.
  - NUMERIC. Use DECIMAL instead.
- **Specifying AS LOCATOR for a parameter:** Passing a locator instead of a value can result in fewer bytes being passed in or out of the function. This can be useful when the value of the parameter is very large. The AS LOCATOR clause specifies that a locator to the value of the parameter is passed instead of the actual value. Specify AS LOCATOR only for parameters that have a LOB or XML data type or a distinct type based on a LOB or XML data type and only when LANGUAGE JAVA is not in effect.

  The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

  AS LOCATOR cannot be specified for SQL functions.

Determining the uniqueness of functions in a schema: The same name can be used for more than one function in a schema if the function signature of each

function is unique. The function signature is the qualified function name combined with the number and data types of the input parameters. The combination of name, schema name, the number of parameters, and the data type each parameter (without regard for other attributes such as length, precision, scale, or CCSID) must not identify a user-defined function that exists at the current server. The return type has no impact on the determining uniqueness of a function. Two different schemas can each contain a function with the same name that have the same data types for all of their corresponding data types. However, a schema must not contain two functions with the same name that have the same data types for all of their corresponding data types.

When determining whether corresponding data types match, the database manager does not consider any length, precision, or scale attributes in the comparison. The database manager considers the synonyms of data types a match. For example, REAL and FLOAT, and DOUBLE and FLOAT are considered a match. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), and DECIMAL(4,3). Furthermore, the character and graphic types are considered to be the same. For example, the following are considered to be the same type: CHAR and GRAPHIC, VARCHAR and VARGRAPHIC, and CLOB and DBCLOB. CHAR(13) and GRAPHIC(8) are considered to be the same type. An error is returned if the signature of the function being created is a duplicate of a signature for an existing user-defined function with the same name and schema.

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (INT, CHAR(15) ...
CREATE FUNCTION PART (INTEGER, CHAR(40) ...

CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

Specifying a specific name for a function: When defining multiple functions with the same name and schema (with different parameter lists), it is recommended that a specific name also be specified. The specific name can be used to uniquely identify the function such as when sourcing on this function, dropping the function, or commenting on the function. However, the function cannot be invoked by its specific name.

The specific name is implicitly or explicitly qualified with a schema name. If a schema name is not specified on CREATE FUNCTION, it is the same as the explicit or implicit schema name of the function name (*function-name*). If a schema name is specified, it must be the same as the explicit or implicit schema name of the function name. The name, including the schema name must not identify the specific name of another function or procedure that exists at the current server.

If a specific name is not specified, it is set to the function name. If a function or procedure with that specific name already exists, a unique name is generated similar to the rules used to generate unique table names.

Extending or overriding a built-in function:

Giving a user-defined function the same name as a built-in function is not a recommended practice unless the functionality of the built-in function needs to be extended or overridden.

- **Extending the functionality of existing built-in functions:**

Create the new user-defined function with the same name as the built-in function, and a unique function signature. For example, a user-defined function similar to the built-in function ROUND that accepts the distinct type MONEY as input rather than the built-in numeric types might be necessary. In this case, the signature for the new user-defined function named ROUND is different from all the function signatures supported by the built-in ROUND function.

- **Overriding a built-in function:**

  Create the new user-defined function with the same name and signature as an existing built-in function. The new function has the same name and data type as the corresponding parameters of the built-in function but implements different logic. For example, a user-defined function similar to the built-in function ROUND that uses different rules for rounding than the built-in ROUND function might be necessary. In this case, the signature for the new user-defined function named ROUND will be the same as a signature that is supported by the built-in ROUND function.

  Once a built-in function has been overridden, if the schema for the new function appears in the SQL path before the system schemas, the database manager may choose a user-defined function rather than the built-in function. An application that uses the unqualified function name and was previously successful using the built-in function of that name might fail, or perhaps even worse, appear to run successfully but provide a different result if the user-defined function is chosen by the database manager rather than the built-in function.

  The DISTINCT keyword can be passed on the invocation of a user-defined function that is sourced on one of the built-in aggregate functions. For example, assume that MY_AVG is a user-defined function that is sourced on the built-in AVG function. The user-defined function could be invoked with `MY_AVG (DISTINCT expression)`. This results in the underlying built-in AVG function being invoked with the DISTINCT keyword.

Special registers in functions: The settings of the special registers of the invoker are inherited by the function on invocation and restored upon return to the invoker. Special registers may be changed in a function that can execute SQL statements, but these changes do not affect the caller.

MODIFIES SQL DATA and EXTERNAL ACTION functions: If a MODIFIES SQL DATA or EXTERNAL ACTION function is invoked in other than the outermost select list, the results are unpredictable since the number of times the function is invoked will vary depending on the access plan used.

Functions and adopted authority: Fenced functions run in separate threads. If ALLOW PARALLEL is specified for a NOT FENCED function, that function may also run in a separate thread. Functions that run in separate threads will not run under any adopted authority that might be specified by the invoking application.

Restore considerations: When a function's associated program or service program is saved and subsequently restored and the object was updated with the function attributes when the function was created, the saved attributes will be processed and possibly changed during the restore.

If the 'Saved library' (SAVLIB) of the program or service program is different from the 'Restore to library' (RSTLIB), the function's schema name, specific schema name, and the external name may be changed as a result of the restore.

- If the saved function schema name and the library name of the saved object match, the function schema will be changed to the 'Restore to library' (RSTLIB). Otherwise, the function schema name is the saved function schema name.

- The specific schema name is always the same as function schema name.
- If the saved EXTERNAL NAME library and the library name of the saved object match, the EXTERNAL NAME library will be changed to the 'Restore to library' (RSTLIB). Otherwise, the EXTERNAL NAME library is the saved library name. If the saved EXTERNAL NAME library is *LIBL, it will not change.

If the same function signature already exists in the catalog:

- If the external program name or service program name is the same as the one that already exists in the catalog, the information in the catalog for that procedure will be replaced with the saved attributes (including the specific name).
- Otherwise, the saved attributes are not restored, and a warning (SQL9015) is issued.

If the same specific name already exists in the catalog, a warning is issued and a new specific name is generated. Otherwise, the specific name of the function is preserved.

CREATE FUNCTION (External Scalar)

This CREATE FUNCTION (External Scalar) statement defines an external scalar function at the current server. A user-defined external scalar function returns a single value each time it is invoked.

Invocation

This statement can be embedded in an application program, or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization id of the statement must include at least one of the following:

- For the SYSFUNCS catalog view and SYSPARMS catalog table[88]:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative Authority

If the external program or service program exists, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the external program or service program that is referenced in the SQL statement:
  - The system authority *EXECUTE on the library that contains the external program or service program.
  - The system authority *EXECUTE on the external program or service program, and
  - The system authority *CHANGE on the program or service program. The system needs this authority to update the program object to contain the information necessary to save/restore the function to another system. If user does not have this authority, the function is still created, but the program object is not updated.
- Administrative Authority

If SQL names are specified and a user profile exists that has the same name as the library into which the function is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

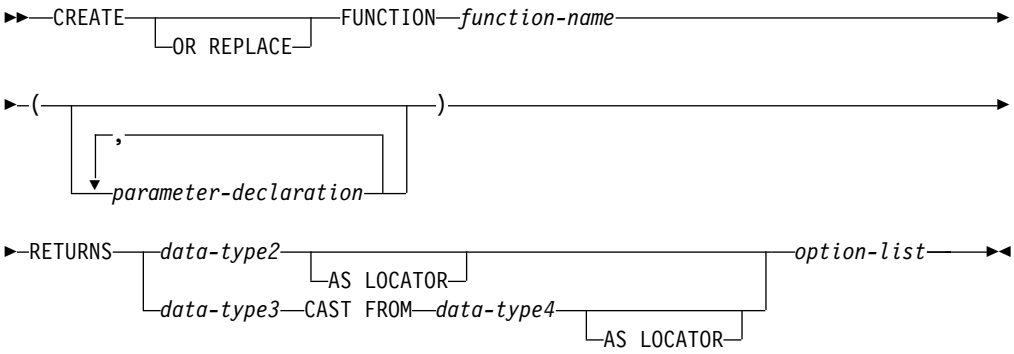
88. The GRTOBJAUT CL command must be used to grant these privileges.

To replace an existing function, the privileges held by the authorization ID of the statement must include at least one of the following:

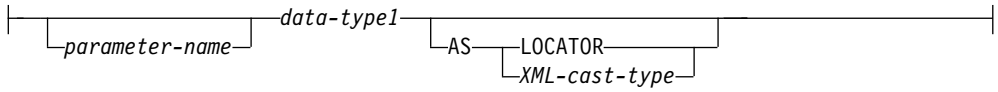
- The following system authorities:
  - The system authority of *OBJMGT on the service program object associated with the function
  - All authorities needed to DROP the function
  - The system authority *READ to the SYSFUNCS catalog view and SYSPARMS catalog table
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax

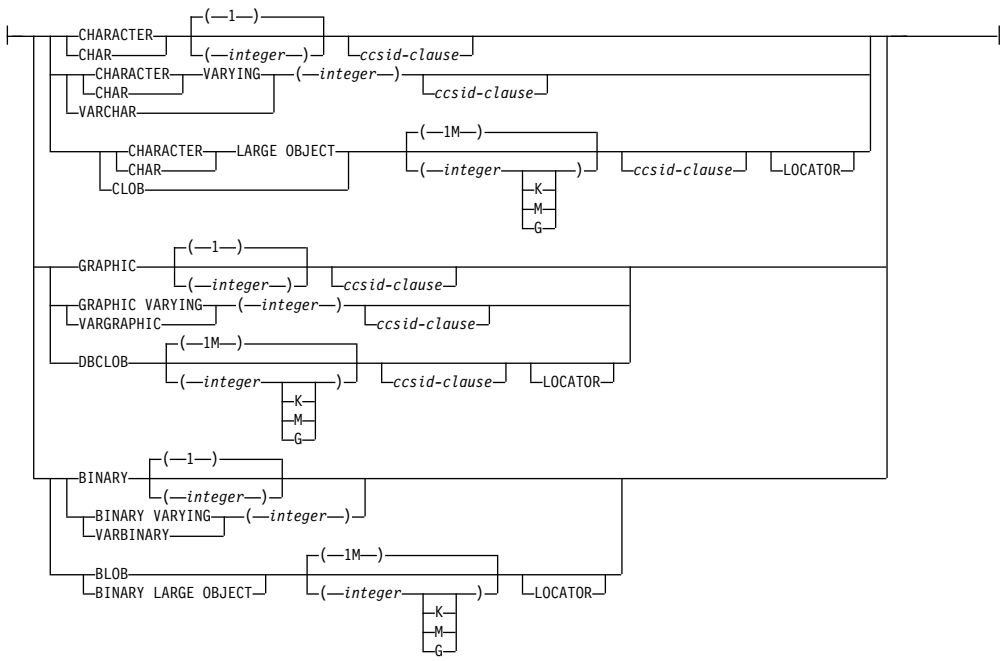


parameter-declaration:

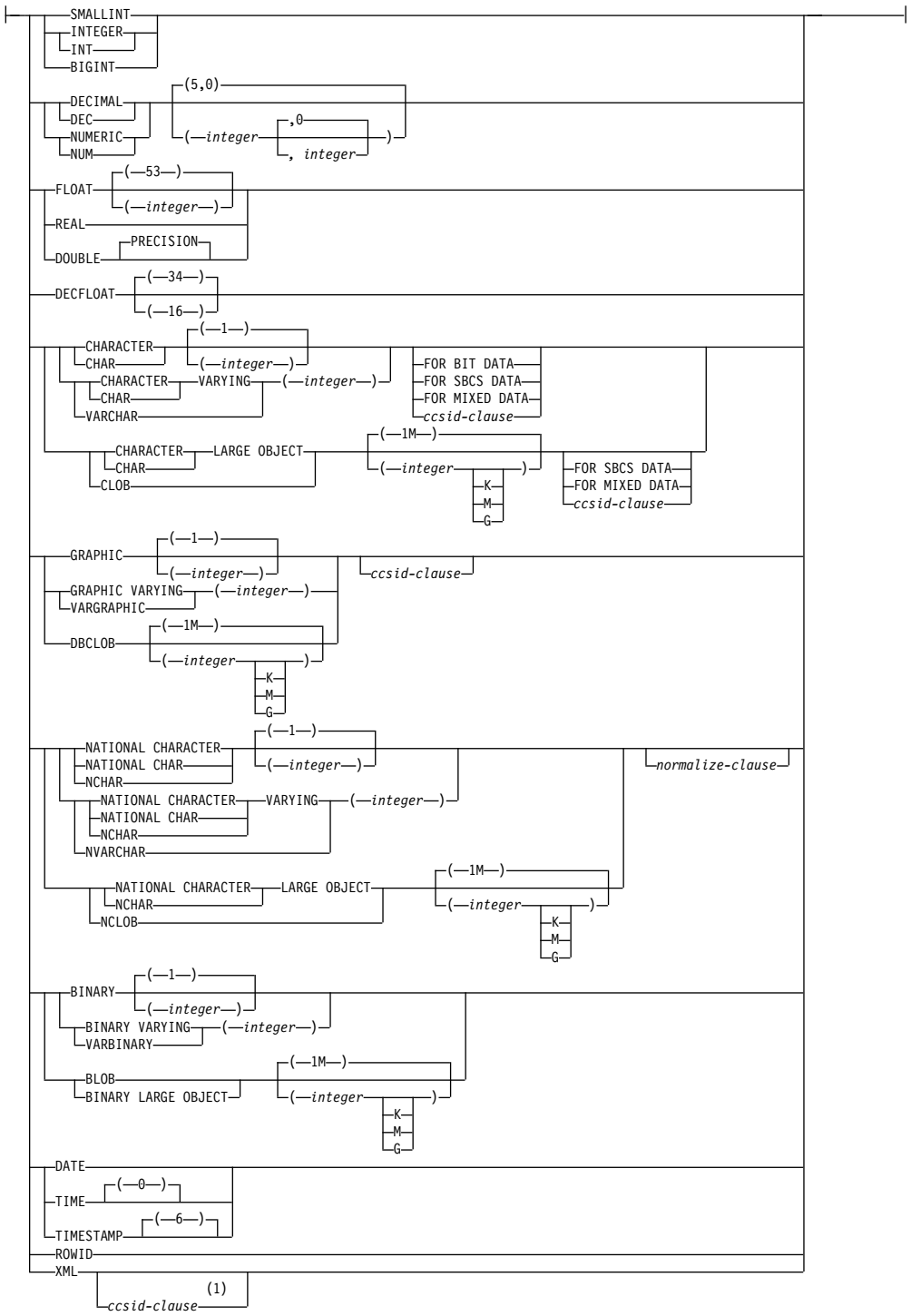


data-type1, data-type2, data-type3, data-type4:

XML-cast-type:



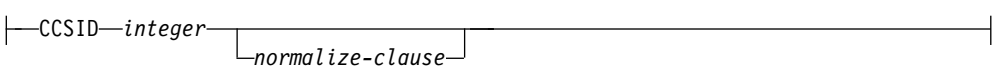
built-in-type:



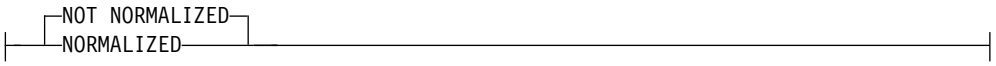
Notes:

1 The *ccsid-clause* for XML is only allowed for *data-type2* and *data-type3*.

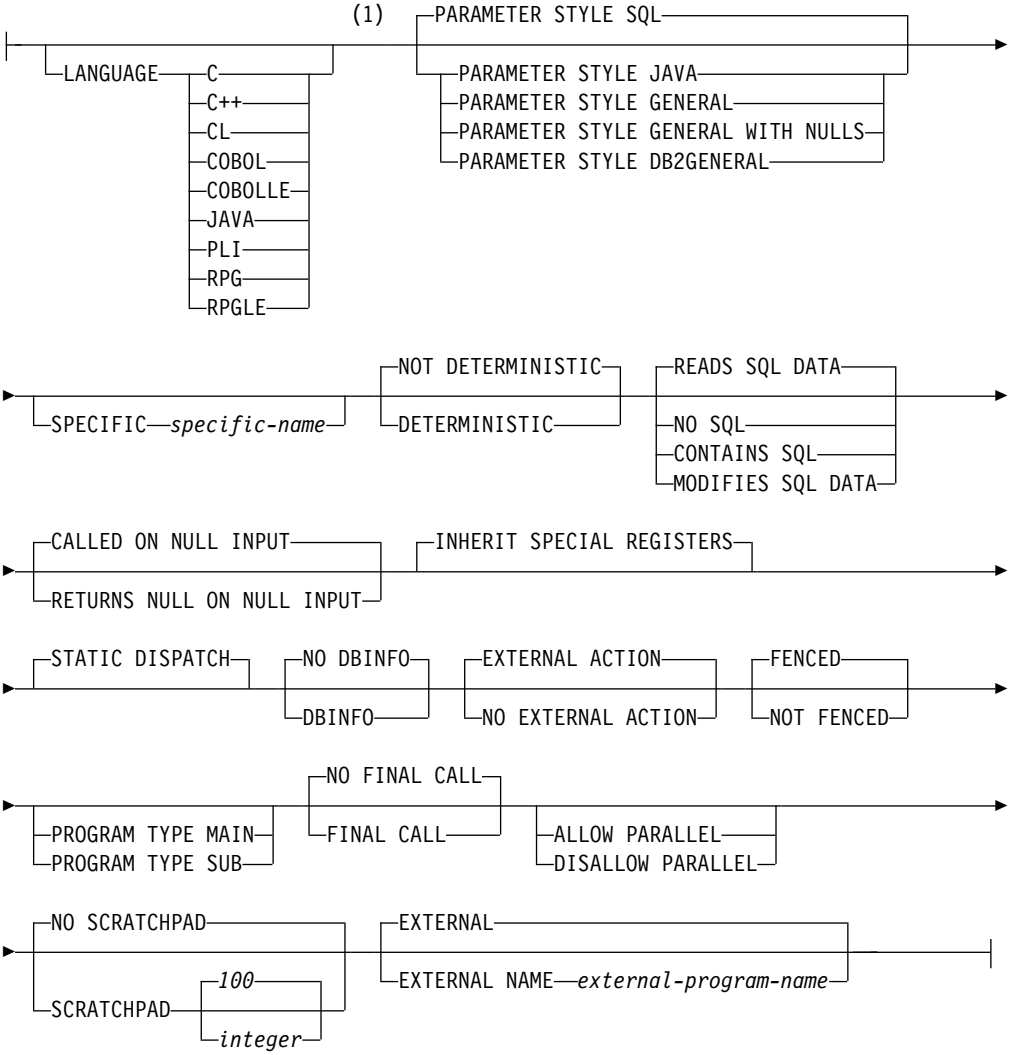
ccsid-clause:



normalize-clause:



option-list:



Notes:

1 This clause and the clauses that follow in the option-list can be specified in any order. Each clause can be specified at most once.

Description

OR REPLACE
Specifies to replace the definition for the function if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog with the exception that privileges that were granted on the function are not affected. This option is ignored if a definition for the function does not exist at the current server. To replace an existing function, the *specific-name* and *function-name* of the new definition must be the same as

the *specific-name* and *function-name* of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new function is created.

function-name
Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in the schema that is specified by the qualifier. If no qualifier is specified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the function will be created in the current library (*CURLIB).
- Otherwise, the function will be created in the current schema.

In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see Choosing the Schema and Function Name.

(*parameter-declaration,...)*
Specifies the number of input parameters of the function and the data type of each parameter. Each *parameter-declaration* specifies an input parameter for the function. A maximum of 90 parameters can be specified. A function can have zero or more input parameters. There must be one entry in the list for each parameter that the function expects to receive. All the parameters for a function are input parameters and are nullable. In the case of JAVA, numeric parameters other than the DECIMAL and NUMERIC types are not nullable. A runtime error will occur if a null value is input to such a parameter for a CALLED ON NULL INPUT function. For more information, see Defining the parameters.

parameter-name
Names the parameter. Although not required, a parameter name can be specified for each parameter. The name cannot be the same as any other *parameter-name* in the parameter list.

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see “CREATE TABLE” on page 982. Some data types are not supported in all languages. For details on the mapping between the SQL data types and host language data types, see Embedded SQL Programming topic collection. Built-in data type specifications can be specified if they correspond to the language that is used to write the user-defined function.

distinct-type-name
Specifies a user-defined distinct type. The length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). For more information on creating a distinct type, see “CREATE TYPE (Distinct)” on page 1055.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

Any parameter that has an XML type must specify either the *XML-cast-type* clause or the AS LOCATOR clause.

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB or XML data type or a distinct type based on a LOB or XML data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

AS *XML-cast-type*
Specifies the data type passed to the function for a parameter that is XML type or a distinct type based on XML type. If LOCATOR is specified, the parameter is a locator to the value rather than the actual value.

If a CCSID value is specified, only Unicode CCSID values can be specified for graphic data types. If a CCSID value is not specified, the CCSID is established at the time the function is created according to the SQL_XML_DATA_CCSID QAQQINI option setting. The default CCSID is 1208. See "XML Values" on page 88 for a description of this option.

RETURNS
Specifies the data type for the result of the function. Consider this clause in conjunction with the optional CAST FROM clause.

data-type2
Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, LONG VARGRAPHIC, or DataLink) or a distinct type (that is not based on a DataLink).

If a CCSID is specified,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in that CCSID.
- If AS LOCATOR is specified and the CCSID of the data the locator points to is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified and the function is not referenced in the outermost select list of a view,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in the CCSID of the job (or associated graphic CCSID of the job for graphic string return values).
- If AS LOCATOR is specified, the data the locator points to is converted to the CCSID of the job, if the CCSID of the data the locator points to is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is

especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

If a CCSID is not specified and the function is referenced in the outermost select list of a view,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in the CCSID of the associated view column.
- If AS LOCATOR is specified, the data the locator points to is converted to the CCSID of the associated view column, if the CCSID of the data the locator points to is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

AS LOCATOR
Specifies that the function returns a locator to the value rather than the actual value. Specify AS LOCATOR only if the result of the function has a LOB or XML data type or a distinct type based on a LOB or XML data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

data-type3 **CAST FROM** *data-type4*
Specifies the data type and attributes of the function (*data-type4*) and the data type in which that result is returned to the invoking statement (*data-type3*). The two data types can be different. For example, for the following definition, the function returns a DOUBLE value, which the database manager converts to a DECIMAL value and then passes to the statement that invoked the function:

```
CREATE FUNCTION SQRT (DECIMAL15,0))
  RETURNS DECIMAL(15,0)
  CAST FROM DOUBLE
  ...
```

The value of *data-type4* must not be XML or a distinct type and must be castable to *data-type3*. The value for *data-type3* can be any built-in data type or distinct type. (For information on casting data types, see "Casting between data types" on page 95).

For CCSID information, see the preceding description of *data-type2*.

AS LOCATOR
Specifies that the function returns a locator to the value rather than the actual value. Specify AS LOCATOR only if the result of the function has a LOB data type or a distinct type based on a LOB data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

LANGUAGE
Specifies the language interface convention to which the function body is written. All programs must be designed to run in the server's environment.

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program at the time the function is created. The language of the program is assumed to be C if:

- The program attribute information associated with the program does not identify a recognizable language
- The program cannot be found

C
The external program is written in C.

C++
The external program is written in C++.

CL
The external program is written in CL or ILE CL.

COBOL
The external program is written in COBOL.

COBOLLE
The external program is written in ILE COBOL.

JAVA
The external program is written in JAVA. The database manager will call the user-defined function, which must be a public static method of the specified Java class

When LANGUAGE JAVA is specified, specify the EXTERNAL NAME clause with a valid *external-java-routine-name*. Do not specify LANGUAGE JAVA when SCRATCHPAD, FINAL CALL, or DBINFO is specified.

PLI
The external program is written in PL/I.

RPG
The external program is written in RPG.

RPGLE
The external program is written in ILE RPG.

PARAMETER STYLE
Specifies the conventions used for passing parameters to and returning the values from functions:

SQL
All applicable parameters are passed. The parameters are defined to be in the following order:

- *n* parameters for the input parameters that are specified for the function.
- A parameter for the result of the function.
- *n* parameters for indicator variables for the input parameters.
- A parameter for the indicator variable for the result.
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the function. The SQLSTATE returned can either be:
  - the SQLSTATE from the last SQL statement executed in the external program,
  - an SQLSTATE that is assigned by the external program.

  The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the function.

- A VARCHAR(517) input parameter for the fully qualified function name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(1000) output parameter for the message text.

  When control is returned to the invoking program, the message text can be found in the 6th token of the SQLERRMC field of the SQLCA. Only a portion of the message text is available. For information on the layout of the message data in the SQLERRMC, see the replacement data descriptions for message SQL0443 in message file QSQLMSG. The complete message text can be retrieved using the GET DIAGNOSTICS statement. For more information, see "GET DIAGNOSTICS" on page 1194.
- Zero to three optional parameters:
  - A structure (consisting of an INTEGER followed by a CHAR(n)) input and output parameter for the scratchpad, if SCRATCHPAD was specified on the CREATE FUNCTION statement.
  - An INTEGER input parameter for the call type, if FINAL CALL was specified on the CREATE FUNCTION statement.
  - A structure for the dbinfo structure, if DBINFO was specified on the CREATE FUNCTION statement.

These parameters are passed according to the specified LANGUAGE. For example, if the language is C or C++, the VARCHAR parameters are passed as NUL-terminated strings. For more information about the parameters passed, see the include sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

DB2GENERAL

This parameter style is used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a Java class. All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- A parameter for the result of the function.

DB2GENERAL is only allowed when the LANGUAGE is JAVA.

GENERAL

All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.

Note that the result is returned as a value of a C value returning function. For example:

```
return_val func(parameter-1, parameter-2, ...)
```

GENERAL is only allowed when EXTERNAL NAME identifies a service program.

GENERAL WITH NULLS

All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- An additional argument is passed for an indicator variable array.

- A parameter for the indicator variable for the result.

Note that the result is returned as a value of a C value returning function. For example:

```
return_val func(parameter-1, parameter-2, ...)
```

GENERAL WITH NULLS is only allowed when EXTERNAL NAME identifies a service program.

JAVA
Specifies that the function will use a parameter passing convention that conforms to the Java language and ISO/IEC FCD 9075-13:2003, *Information technology - Database languages - SQL - Part 13: Java Routines and Types (SQL/JRT)* specification. All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.

Note that the result is returned as a value of a C value returning function. For example:

```
return_val func(parameter-1, parameter-2, ...)
```

JAVA is only allowed when the LANGUAGE is JAVA.

Note that the language of the external function determines how the parameters are passed. For example, in C, any VARCHAR or CHAR parameters are passed as NUL-terminated strings. For more information, see the SQL Programming topic collection. For Java routines, see the IBM Developer Kit for Java topic collection.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a function.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. The database manager uses this information during optimization of SQL statements. An example of a function that is not deterministic is one that generates random numbers.

A function that is not deterministic might return incorrect results if the function is executed by parallel tasks. Specify the DISALLOW PARALLEL clause for these functions.

NOT DETERMINISTIC should be specified if the function contains a reference to a special register, a non-deterministic function, or a sequence.

DETERMINISTIC
Specifies that the function always returns the same result each time that the function is invoked with the same input arguments. The database

manager uses this information during optimization of SQL statements.[89] An example of a deterministic function is a function that calculates the square root of the input argument.

CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA, or NO SQL
Specifies the classification of SQL statements that the function can execute. The database manager verifies that the SQL statements that the function issues are consistent with this specification. For the classification of each statement, see Appendix B, "Characteristics of SQL statements," on page 1501. The default is READS SQL DATA.

READS SQL DATA
Specifies that the function can execute statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

NO SQL
Specifies that the function can execute only SQL statements with a data access classification of NO SQL.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access classification of CONTAINS SQL or NO SQL. The function cannot execute any SQL statements that read or modify data.

MODIFIES SQL DATA
The function can execute any SQL statement except those statements that are not supported in any function.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
Specifies whether the function is called if any of the input arguments is null at execution time. CALLED ON NULL INPUT is the default.

RETURNS NULL ON INPUT
Specifies that the function is not invoked if any of the input arguments is null. The result is the null value.

CALLED ON NULL INPUT
Specifies that the function is to be invoked, if any, or all, argument values are null. This specification means that the function must be coded to test for null argument values. The function can return a null or nonnull value.

INHERIT SPECIAL REGISTERS
Specifies that existing values of special registers are inherited upon entry to the function.

STATIC DISPATCH
Specifies that the function is dispatched statically. All functions are statically dispatched.

NO DBINFO or DBINFO
Specifies whether additional status information is passed when the function is invoked. The default is NO DBINFO.

NO DBINFO
Specifies that no additional information is passed.

89. The query optimizer may choose to cache deterministic scalar function results. The DETERMINISTIC_UDF_SCOPE QAQQINI option can be used to control the scope of the caching. If the result of the function contains sensitive data, consider using the DETERMINISTIC_UDF_SCOPE QAQQINI option or create the function NOT DETERMINISTIC to prevent inadvertent access to the result. For more information, see the Database Performance and Query Optimization topic collection.

DBINFO

Specifies that the database manager should pass a structure containing status information to the function. Table 74 contains a description of the DBINFO structure. Detailed information about the DBINFO structure can be found in include sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

DBINFO is only allowed with PARAMETER STYLE SQL or PARAMETER STYLE DB2GENERAL.

Table 74. DBINFO fields

Field	Data Type	Description
Relational database	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The run-time authorization ID.
CCSID Information	INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER CHAR(8)	The CCSID information of the job. Three sets of three CCSIDs are returned. The following information identifies the three CCSIDs in each set: • SBCS CCSID • DBCS CCSID • Mixed CCSID Following the three sets of CCSIDs is an integer that indicates which set of three sets of CCSIDs is applicable and eight bytes of reserved space. Each set of CCSIDs is for a different encoding scheme (EBCDIC, ASCII, and Unicode). If a CCSID is not explicitly specified for a parameter on the CREATE FUNCTION statement, the input string is assumed to be encoded in the CCSID of the job at the time the function is executed. If the CCSID of the input string is not the same as the CCSID of the parameter, the input string passed to the external function will be converted before calling the external program.
Target column	VARCHAR(128) VARCHAR(128) VARCHAR(128)	If a user-defined function is specified on the right-hand side of a SET clause in an UPDATE statement, the following information identifies the target column: • Schema name • Base table name • Column name If the user-defined function is not on the right-hand side of a SET clause in an UPDATE statement, these fields are blank.
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server's platform type.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a stream file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function can take an action that changes the state of an object that the database manager does not manage. Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action.

NO EXTERNAL ACTION
The function does not perform an external action. It need not be called with each successive function invocation.

NO EXTERNAL ACTION functions might perform better than EXTERNAL ACTION functions because they might not be invoked for each successive function invocation.

FENCED or NOT FENCED
Specifies whether the external function runs in an environment that is isolated from the database manager environment. FENCED is the default.

FENCED
The function will run in a separate thread.

FENCED functions cannot keep SQL cursors open across individual calls to the function. However, the cursors in one thread are independent of the cursors in any other threads which reduces the possibility of cursor name conflicts.

NOT FENCED
The function may run in the same thread as the invoking SQL statement.

NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will also be preserved between calls to the function. However, cursor names may conflict since the UDF is now running in the same thread as the invoking SQL statement and other NOT FENCED UDFs.

NOT FENCED functions usually perform better than FENCED functions.

PROGRAM TYPE MAIN or PROGRAM TYPE SUB
This parameter is allowed for compatibility with other products. It indicates whether the routine's external program is a program (*PGM) or a procedure in a service program (*SRVPGM).

PROGRAM TYPE MAIN
Specifies that the routine executes as the main entry point in a program. The external program must be a *PGM object.

PROGRAM TYPE SUB
Specifies that the routine executes as a procedure in a service program. The external program must be a *SRVPGM object.

NO FINAL CALL or FINAL CALL
Specifies whether a *final call* is made to the function. A final call enables the function to free any system resources that it has acquired. A final call is useful when the function has been defined with the SCRATCHPAD keyword and the function acquires system resources and stores them in the scratchpad. The default is NO FINAL CALL.

NO FINAL CALL
Specifies that a final call is not made to the function. The function does not receive an additional argument that specifies the type of call.

FINAL CALL
Specifies that a final call is made to the function. To differentiate between final calls and other calls, the function receives an additional argument that specifies the type of call.

FINAL CALL is only allowed with PARAMETER STYLE SQL or PARAMETER STYLE DB2GENERAL.

The types of calls are:

First Call
Specifies the first call to the function for this reference to the function in this SQL statement. A first call is a normal call. SQL arguments are passed and the function is expected to return a result.

Normal Call
Specifies that SQL arguments are passed and the function is expected to return a result.

Final Call
Specifies the last call to the function to enable the function to free resources. A final call is not a normal call. If an error occurs, the database manager attempts to make the final call.

A final call occurs at these times:

- *End of statement:* When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
- *End of a parallel task:* When the function is executed by parallel tasks.
- *End of transaction:* When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

Some functions that use a final call can receive incorrect results if parallel tasks execute the function. For example, if a function sends a note for each final call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that have inappropriate actions when executed in parallel.

If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made when the cursor is closed or the application ends. If a commit occurs at the end of a parallel task, a final call is made regardless of whether a cursor defined as WITH HOLD is open.

Committable operations should not be performed during a FINAL CALL, because the FINAL CALL may occur during a close that is invoked as part of a COMMIT operation.

ALLOW PARALLEL or DISALLOW PARALLEL
Specifies whether the function can be run in parallel.

The default is DISALLOW PARALLEL if one or more of the following clauses are specified: NOT DETERMINISTIC, EXTERNAL ACTION, FINAL CALL, MODIFIES SQL DATA, or SCRATCHPAD. Otherwise, ALLOW PARALLEL is the default.

ALLOW PARALLEL
Specifies that the database manager can consider parallelism for the

function. The database manager is not required to use parallelism on the SQL statement that invokes the function or on any SQL statement issued from within the function.

See the descriptions of NOT DETERMINISTIC, EXTERNAL ACTION, MODIFIES SQL DATA, SCRATCHPAD, and FINAL CALL for considerations that apply to specification of ALLOW PARALLEL.

DISALLOW PARALLEL
Specifies that the database manager must not use parallelism for the function.

NO SCRATCHPAD or SCRATCHPAD
Specifies whether the function requires a static memory area.

NO SCRATCHPAD
Specifies that the function does not require a persistent memory area.

SCRATCHPAD *integer*
Specifies that the function requires a persistent memory area of length integer. The integer can range from 1 to 16,000,000. If the memory area is not specified, the size of the area is 100 bytes. If parameter style DB2SQL is specified, a pointer is passed following the required parameters that points to a static storage area. If ALLOW PARALLEL is specified, a memory area is allocated for each user-defined function reference in the statement. If DISALLOW PARALLEL is specified, only 1 memory area will be allocated for the function.

The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, three scratchpads are allocated for the three references to UDFX in the following SQL statement:

```
SELECT A, UDFX(A)
  FROM TABLEB
  WHERE UDFX(A) > 103 OR UDFX(A) < 19
```

If the function is run under parallel tasks, one scratchpad is allocated for each parallel task of each reference to the function in the SQL statement. This can lead to unpredictable results. For example, if a function uses the scratchpad to count the number of times that it is invoked, the count reflects the number of invocations done by the parallel task and not the SQL statement. Specify the DISALLOW PARALLEL clause for functions that will not work correctly with parallelism.

SCRATCHPAD is only allowed with PARAMETER STYLE SQL or PARAMETER STYLE DB2GENERAL.

EXTERNAL
Specifies that the CREATE FUNCTION statement is being used to define a new function that is based on code that is written in an external programming language.

If *external-program-name* is not specified, the external program name is assumed to be the same as the function name.

NAME *external-program-name*
Specifies the program, service program, or Java class that will be executed when the function is invoked in an SQL statement. The name must identify

a program, service program, or Java class that exists at the application server at the time the function is invoked. If the naming option is *SYS and the name is not qualified:

- The current path will be used to search for the program at the time the function is invoked.
- *LIBL will be used to search for the program or service program at the time COMMENT, GRANT, LABEL, or REVOKE operations are performed on the function.

The validity of the name is checked at the application server. If the format of the name is not correct, an error is returned.

The program, service program, or Java class need not exist at the time the function is created, but it must exist at the time the function is invoked.

CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK, and SET TRANSACTION statements are not allowed in the external program of the function.

Notes

General considerations for defining user-defined functions: See "CREATE FUNCTION" on page 851 for general information on defining user-defined functions.

REPLACE rules: When an external function is recreated by REPLACE:

- Any existing comment or label is discarded.
- If a different external program is specified:
  - Authorized users are not copied to the new program.
  - Journal auditing is not changed.
- Otherwise:
  - Authorized users are maintained. The object owner will not change.
  - Current journal auditing is not changed.

Creating the function: When an external function associated with an ILE external program or service program is created, an attempt is made to save the function's attributes in the associated program or service program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the attributes are used to update the catalogs.

The attributes can be saved for external functions subject to the following restrictions:

- The external program library must not be QSYS.
- The external program must exist when the CREATE FUNCTION statement is issued.

  If system naming is specified and the external program name is not qualified, the external program must be found in the library list.
- The external program must be an ILE *PGM or *SRVPGM object.
- The external program must not already contain attributes for 32 routines.

If the object cannot be updated, the function will still be created.

Invoking the function: When an external function is invoked, it runs in whatever activation group was specified when the external program or service program was

created. However, ACTGRP(*CALLER) should normally be used so that the function runs in the same activation group as the calling program. ACTGRP(*NEW) is not allowed.

LANGUAGE JAVA functions always run in the default activation group (*DFTACTGRP). Caution should be used when writing MODIFIES SQL DATA Java functions. Since changes performed by the Java function are performed in the default activation group, transaction problems may occur if the invoker runs in a new activation group (*NEW).

Notes for Java functions: To be able to run Java functions, you must have the IBM Developer Kit for Java (5761-JV1) installed on your system. Otherwise, an SQLCODE of -443 will be returned and a CPDB521 message will be placed in the job log.

If an error occurs while running a Java function, an SQLCODE of -443 will be returned. Depending on the error, other messages may exist in the job log of the job where the function was run.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords SIMPLE CALL can be used as a synonym for GENERAL.
- The keyword DB2GENRL may be used as a synonym for DB2GENERAL.
- The value DB2SQL may be used as a synonym for SQL.
- The keywords PARAMETER STYLE in the PARAMETER STYLE clause are optional.
- The keywords IS DETERMINISTIC may be used as a synonym for DETERMINISTIC.

Examples

Example 1: Assume an external function program in C is needed that implements the following logic:

```
rslt   = 2 * input - 4
```

The function should return a null value if and only if one of the input arguments is null. The simplest way to avoid a function call and get a null result when an input value is null is to specify RETURNS NULL ON NULL INPUT on the CREATE FUNCTION statement. The following statement defines the function, using the specific name MINENULL1.

```
CREATE FUNCTION NTEST1 (SMALLINT)
    RETURNS SMALLINT
    EXTERNAL NAME NTESTMOD
    SPECIFIC MINENULL1
    LANGUAGE C
    DETERMINISTIC
    NO SQL
    FENCED
    PARAMETER STYLE SQL
    RETURNS NULL ON NULL INPUT
    NO EXTERNAL ACTION
```

The program code:

```
void nudft1
  (int *input,                /* ptr to input argument         */
   int *output,               /* ptr to output argument        */
   short *input_ind,          /* ptr to input indicator        */
   short *output_ind,         /* ptr to output indicator       */
   char sqlstate[6],  /* sqlstate                      */
   char fname[140],   /* fully qualified function name */
   char finst[129],   /* function specific name        */
   char msgtext[71])  /* msg text buffer               */
{
  if (*input_ind == -1)
    *output_ind = -1;
  else
{
    *output = 2*(*input)-4;
    *output_ind = 0;
  }
  return;
}
```

Example 2: Assume that a user wants to define an external function named CENTER. The function program will be written in C. The following statement defines the function, and lets the database manager generate a specific name for the function. The name of the program containing the function body is the same as the name of the function, so the EXTERNAL clause does not include 'NAME *external-program-name*'.

```
CREATE FUNCTION CENTER (INTEGER, FLOAT)
   RETURNS FLOAT
   LANGUAGE C
   DETERMINISTIC
   NO SQL
   PARAMETER STYLE SQL
   NO EXTERNAL ACTION
```

Example 3: Assume that user McBride (who has administrative authority) wants to define an external function named CENTER in the SMITH schema. McBride plans to give the function specific name FOCUS98. The function program uses a scratchpad to perform some one-time only initialization and save the results. The function program returns a value with a DOUBLE data type. The following statement written by user McBride defines the function and ensures that when the function is invoked, it returns a value with a data type of DECIMAL(8,4).

```
CREATE FUNCTION SMITH.CENTER (DOUBLE, DOUBLE, DOUBLE)
   RETURNS DECIMAL(8,4)
   CAST FROM DOUBLE
   EXTERNAL NAME CMOD
   SPECIFIC FOCUS98
   LANGUAGE C
   DETERMINISTIC
   NO SQL
   FENCED
   PARAMETER STYLE SQL
   NO EXTERNAL ACTION
   SCRATCHPAD
   NO FINAL CALL
```

Example 4: The following example defines a Java user-defined function that returns the position of the first vowel in a string. The user-defined function is written in Java, is to be run fenced, and is the FINDVWL method of class JAVAUDFS.

```
CREATE FUNCTION FINDV (VARCHAR(32000))
   RETURNS INTEGER
   FENCED
```

```
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'JAVAUDFS.FINDVWL'
NO EXTERNAL ACTION
CALLED ON NULL INPUT
DETERMINISTIC
NO SQL
```

CREATE FUNCTION (External Table)

This CREATE FUNCTION (External Table) statement defines an external table function at the current server. The function returns a result table.

An external user-defined *table function* may be used in the FROM clause of a subselect, and returns a table to the subselect by returning one row each time it is invoked.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization id of the statement must include at least one of the following:

- For the SYSFUNCS catalog view and SYSPARMS catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative Authority

If the external program or service program exists, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the external program or service program that is referenced in the SQL statement:
  - The system authority *EXECUTE on the library that contains the external program or service program.
  - The system authority *EXECUTE on the external program or service program, and
  - The system authority *CHANGE on the program or service program. The system needs this authority to update the program object to contain the information necessary to save/restore the function to another system. If user does not have this authority, the function is still created, but the program object is not updated.
- Administrative Authority

If SQL names are specified and a user profile exists that has the same name as the library into which the function is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

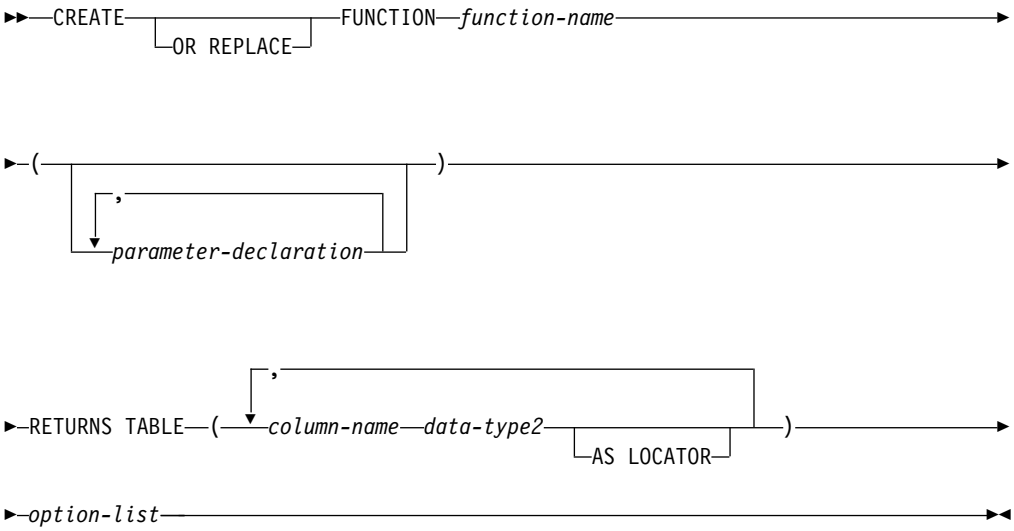
- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

To replace an existing function, the privileges held by the authorization ID of the statement must include at least one of the following:

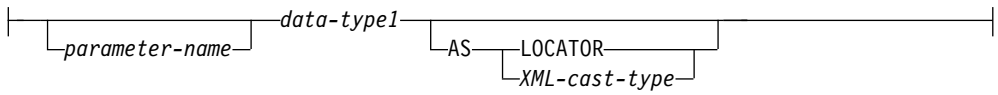
- The following system authorities:
  - The system authority of *OBJMGT on the service program object associated with the function
  - All authorities needed to DROP the function
  - The system authority *READ to the SYSFUNCS catalog view and SYSPARMS catalog table
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View and Corresponding System Authorities When Checking Privileges to a Distinct Type.

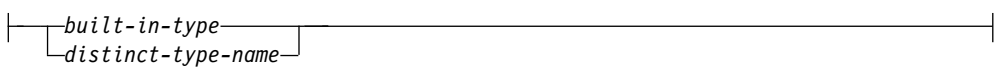
Syntax



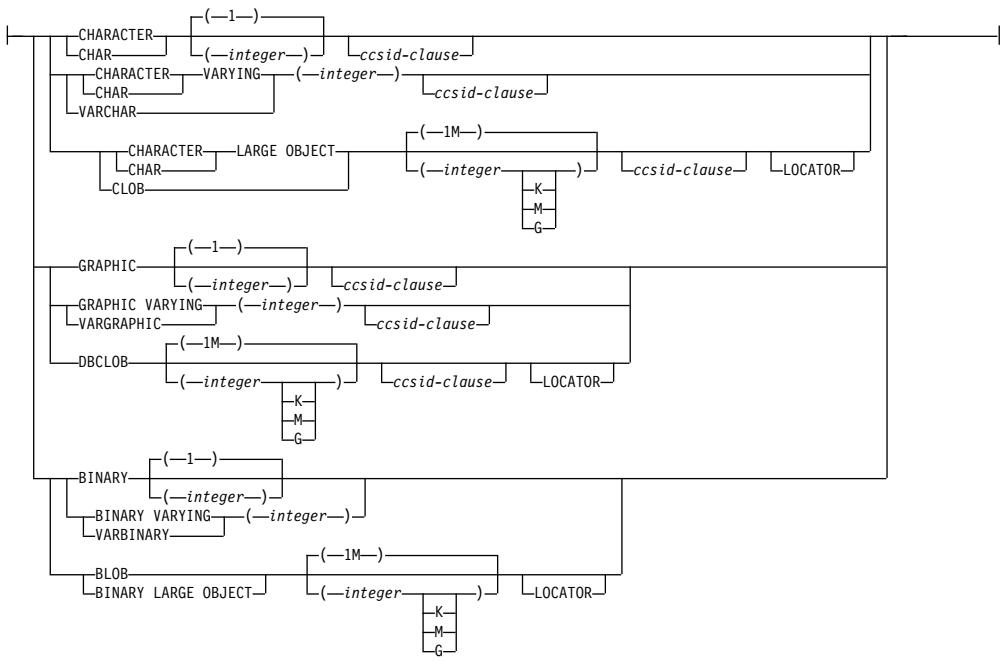
parameter-declaration:



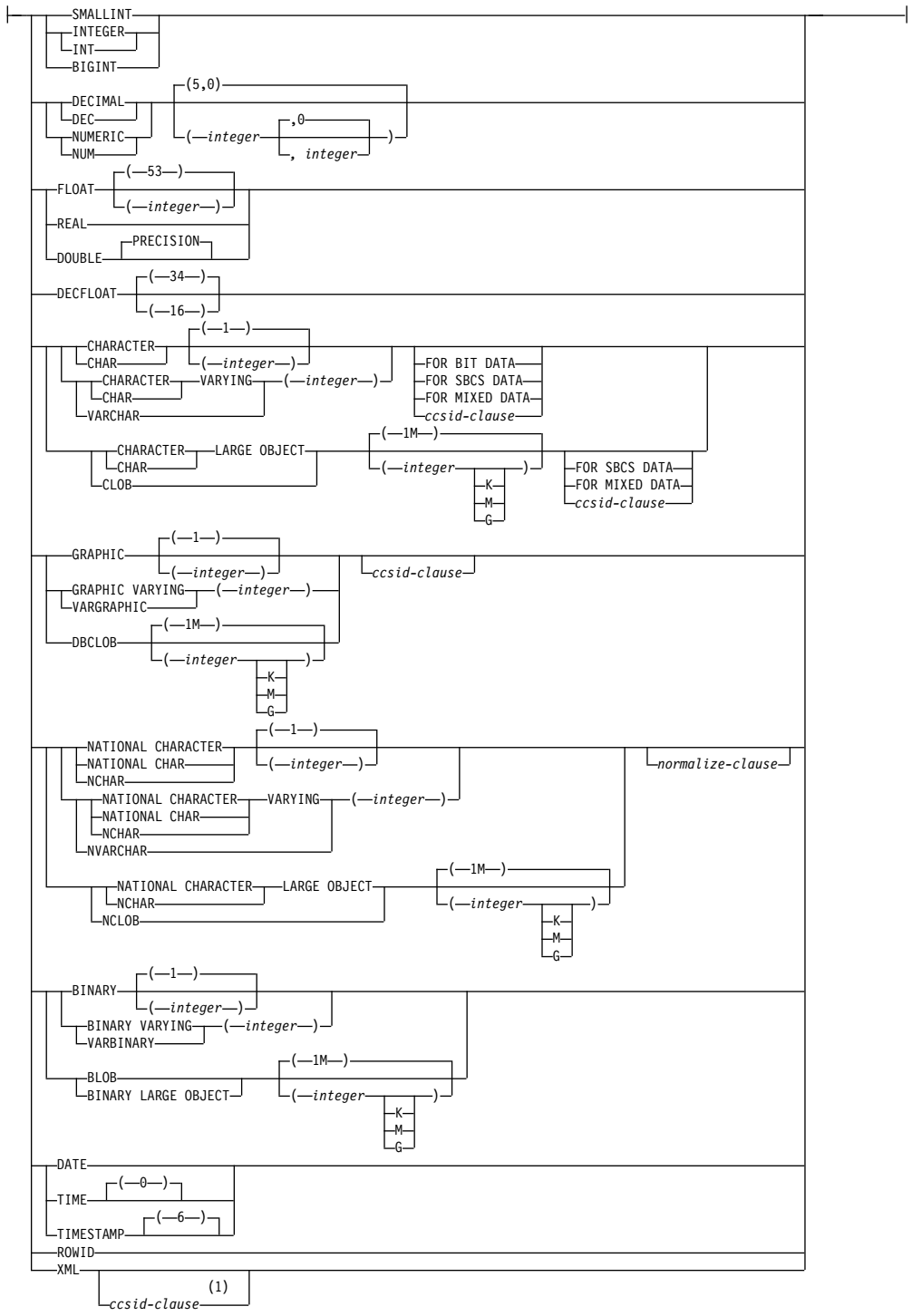
data-type1, data-type2:



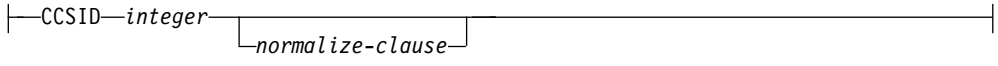
XML-cast-type:



built-in-type:



ccsid-clause:

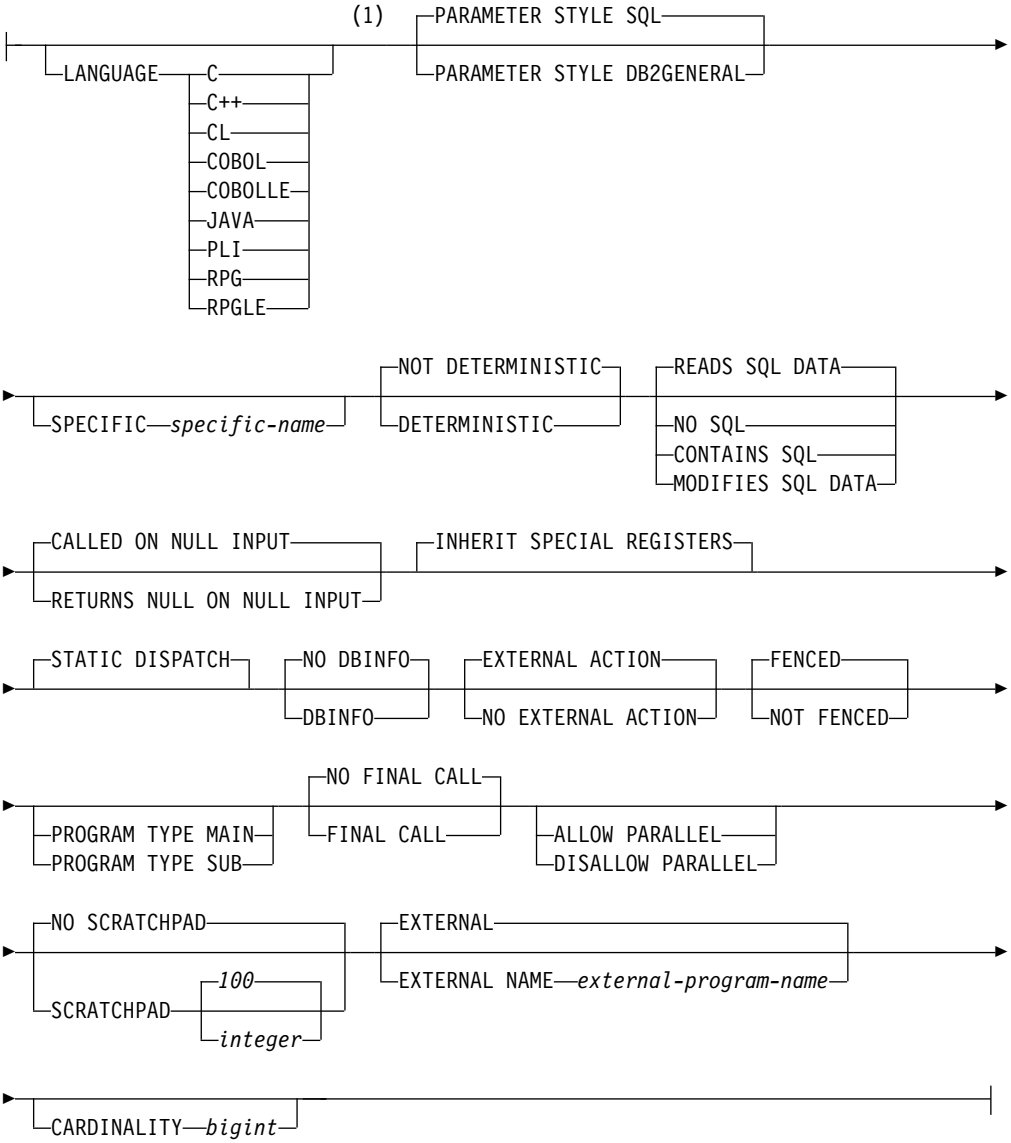


normalize-clause:

Notes:

1 The *ccsid-clause* for XML is only allowed for *data-type2*

option-list:



Notes:

1 This clause and the clauses that follow in the option-list can be specified in any order. Each clause can be specified at most once.

Description

OR REPLACE
Specifies to replace the definition for the function if one exists at the current server. The existing definition is effectively dropped before the new definition

is replaced in the catalog with the exception that privileges that were granted on the function are not affected. This option is ignored if a definition for the function does not exist at the current server. To replace an existing function, the *specific-name* and *function-name* of the new definition must be the same as the *specific-name* and *function-name* of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new function is created.

function-name
Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in the schema that is specified by the qualifier. If no qualifier is specified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the function will be created in the current library (*CURLIB).
- Otherwise, the function will be created in the current schema.

In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see Choosing the Schema and Function Name.

(*parameter-declaration***,...)**
Specifies the number of input parameters of the function and the data type of each parameter. Each *parameter-declaration* specifies an input parameter for the function. A maximum of 90 parameters can be specified. A function can have zero or more input parameters. There must be one entry in the list for each parameter that the function expects to receive. All the parameters for a function are input parameters and are nullable. In the case of JAVA, numeric parameters other than the DECIMAL and NUMERIC types are not nullable. A runtime error will occur if a null value is input to such a parameter for a CALLED ON NULL INPUT function. For more information, see Defining the parameters.

parameter-name
Names the parameter. Although not required, a parameter name can be specified for each parameter. The name cannot be the same as any other *parameter-name* in the parameter list.

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see "CREATE TABLE" on page 982. Some data types are not supported in all languages. For details on the mapping between the SQL data types and host language data types, see Embedded SQL Programming topic collection. Built-in data type specifications can be specified if they correspond to the language that is used to write the user-defined function.

distinct-type-name
Specifies a user-defined distinct type. The length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). For more information about creating a distinct type, see "CREATE TYPE (Distinct)" on page 1055

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

Parameters with a large object (LOB) data type are not supported when PARAMETER STYLE JAVA is specified.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

Any parameter that has an XML type must specify either the *XML-cast-type* clause or the AS LOCATOR clause.

AS LOCATOR
Specifies that the input parameter is a locator to the value rather than the actual value. You can specify AS LOCATOR only if the input parameter has a LOB or XML data type or a distinct type based on a LOB or XML data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

AS *XML-cast-type*
Specifies the data type passed to the function for a parameter that is XML type or a distinct type based on XML type. If LOCATOR is specified, the parameter is a locator to the value rather than the actual value.

If a CCSID value is specified, only Unicode CCSID values can be specified for graphic data types. If a CCSID value is not specified, the CCSID is established at the time the function is created according to the SQL_XML_DATA_CCSID QAQQINI option setting. The default CCSID is 1208. See "XML Values" on page 88 for a description of this option.

RETURNS TABLE
Specifies that the output of the function is a table. The parenthesis that follow this clause enclose a list of names and the data types of the columns of the result table.

Assume that the number of parameters is *n*. For PARAMETER STYLE DB2GENERAL, there must be no more than (255-(*n**2))/2 columns. For PARAMETER STYLE SQL, there must be no more than (247-(*n**2))/2 columns.

column-name
Specifies the name of a column of the output table. Do not specify the same name more than once.

data-type2
Specifies the data type of the column. The column is nullable.

You can specify any built-in data type (except LONG VARCHAR, LONG VARGRAPHIC, or DataLink) or a distinct type (that is not based on a DataLink).

If a DATE or TIME is specified, the table function must return the date or time in ISO format.

If a CCSID is specified,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in that CCSID.
- If AS LOCATOR is specified and the CCSID of the data the locator points to is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified and the function is not referenced in a view,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in the CCSID of the job (or associated graphic CCSID of the job for graphic string return values).
- If AS LOCATOR is specified, the data the locator points to is converted to the CCSID of the job, if the CCSID of the data the locator points to is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

If a CCSID is not specified and the function is referenced in a view,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in the CCSID of the associated view column.
- If AS LOCATOR is specified, the data the locator points to is converted to the CCSID of the associated view column, if the CCSID of the data the locator points to is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

AS LOCATOR
Specifies that the function returns a locator to the value for the column rather than the actual value. You can specify AS LOCATOR only for a LOB or XML data type or a distinct type based on a LOB or XML data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

LANGUAGE
The language clause specifies the language of the external program.

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program at the time the function is created. The language of the program is assumed to be C if:

- The program attribute information associated with the program does not identify a recognizable language
- The program cannot be found

C
The external program is written in C.

C++
The external program is written in C++.

CL
The external program is written in CL or ILE CL.

COBOL
The external program is written in COBOL.

COBOLLE
The external program is written in ILE COBOL.

JAVA
The external program is written in JAVA. The database manager will call the user-defined function as a method in a Java class.

PLI
The external program is written in PL/I.

RPG
The external program is written in RPG.

RPGLE
The external program is written in ILE RPG.

PARAMETER STYLE
Specifies the conventions used for passing parameters to and returning the values from functions:

DB2GENERAL
This parameter style is used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a Java class. All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- The next M parameters are the result columns of the function that are specified on the RETURNS TABLE clause.

DB2GENERAL is only allowed when the LANGUAGE is JAVA.

SQL
All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- The next M parameters are the result columns of the function that are specified on the RETURNS TABLE clause.
- N parameters for indicator variables for the input parameters.
- M parameters for the indicator variables of the result columns of the function that are specified on the RETURNS TABLE clause
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the function. The SQLSTATE returned either be:
  - the SQLSTATE from the last SQL statement executed in the external program,
  - an SQLSTATE that is assigned by the external program.

    The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the function.

- A VARCHAR(517) input parameter for the fully qualified function name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(1000) output parameter for the message text.
- A structure (consisting of an INTEGER followed by a CHAR(n)) input and output parameter for the scratchpad, if SCRATCHPAD was specified on the CREATE FUNCTION statement.
- An INTEGER input parameter for the call type.
- A structure for the dbinfo structure, if DBINFO was specified on the CREATE FUNCTION statement.

These parameters are passed according to the specified LANGUAGE. For example, if the language is C or C++, the VARCHAR parameters are passed as NUL-terminated strings. For more information about the parameters passed, see the include sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

Note that the language of the external function determines how the parameters are passed. For example, in C, any VARCHAR or CHAR parameters are passed as NUL-terminated strings. For more information, see the SQL Programming topic collection. For Java routines, see the IBM Developer Kit for Java topic collection.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a function.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. The database manager uses this information during optimization of SQL statements. An example of a table function that is not deterministic is one that references special registers, non-deterministic functions, or a sequence in a way that affects the table function result table.

DETERMINISTIC
Specifies that the function always returns the same result table each time that the function is invoked with the same input arguments. The database manager uses this information during optimization of SQL statements.[90]

CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA, or NO SQL
Specifies the classification of SQL statements that the function can execute. The database manager verifies that the SQL statements that the function issues are consistent with this specification. For the classification of each statement, see Appendix B, "Characteristics of SQL statements," on page 1501. The default is READS SQL DATA.

READS SQL DATA
Specifies that the function can execute statements with a data access

90. The query optimizer may choose to cache deterministic table function results. The DETERMINISTIC_UDF_SCOPE QAQQINI option can be used to control the scope of the caching. For more information, see the Database Performance and Query Optimization topic collection.

classification of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

NO SQL
Specifies that the function can execute only SQL statements with a data access classification of NO SQL.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access classification of CONTAINS SQL or NO SQL. The function cannot execute any SQL statements that read or modify data.

MODIFIES SQL DATA
The function can execute any SQL statement except those statements that are not supported in any function.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
Specifies whether the function is called if any of the input arguments is null at execution time. CALLED ON NULL INPUT is the default.

RETURNS NULL ON NULL INPUT
Specifies that the function is not called if any of the input arguments is null. The result is an empty table, which is a table with no rows.

CALLED ON NULL INPUT
Specifies that the function is to be invoked, if any argument values are null. This specification means that the function must be coded to test for null argument values. The function can return an empty table, depending on its logic.

INHERIT SPECIAL REGISTERS
Specifies that existing values of special registers are inherited upon entry to the function.

STATIC DISPATCH
Specifies that the function is dispatched statically. All functions are statically dispatched.

NO DBINFO or DBINFO
Specifies whether additional status information is passed when the function is invoked. The default is NO DBINFO.

NO DBINFO
Specifies that no additional information is passed.

DBINFO
Specifies that the database manager should pass a structure containing status information to the function. Table 75 contains a description of the DBINFO structure. Detailed information about the DBINFO structure can be found in sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

Table 75. DBINFO fields

Field	Data Type	Description
Relational database	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The run-time authorization ID.

Table 75. DBINFO fields (continued)

Field	Data Type	Description
CCSID Information	INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER CHAR(8)	The CCSID information of the job. Three sets of three CCSIDs are returned. The following information identifies the three CCSIDs in each set: • SBCS CCSID • DBCS CCSID • Mixed CCSID Following the three sets of CCSIDs is an integer that indicates which set of three sets of CCSIDs is applicable and eight bytes of reserved space. Each set of CCSIDs is for a different encoding scheme (EBCDIC, ASCII, and Unicode). If a CCSID is not explicitly specified for a parameter on the CREATE FUNCTION statement, the input string is assumed to be encoded in the CCSID of the job at the time the function is executed. If the CCSID of the input string is not the same as the CCSID of the parameter, the input string passed to the external function will be converted before calling the external program.
Target column	VARCHAR(128) VARCHAR(128) VARCHAR(128)	If a user-defined function is specified on the right-hand side of a SET clause in an UPDATE statement, the following information identifies the target column: • Schema name • Base table name • Column name If the user-defined function is not on the right-hand side of a SET clause in an UPDATE statement, these fields are blank.
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server's platform type.
Number of table function column list entries	SMALLINT	The number of non-zero entries in the table function column list specified in the "Table function column list" field below.
Reserved	CHAR(24)	Reserved for future use.

Table 75. DBINFO fields (continued)

Field	Data Type	Description
Table function column list	Pointer (16 Bytes)	This field is a pointer to an array of short integers which is dynamically allocated by the database manager. Only the first n entries, where n is specified in the "Number of table function column list entries" field, are of interest, n may be equal to 0, and is less than or equal to the number of result columns defined for the function in the RETURNS TABLE clause. The values correspond to the ordinal numbers of the columns which this statement needs from the table function. A value of 1 means the first defined result column, 2 means the second defined result column, and so on. The values may be in any order. Note that n could be equal to zero for a statement that is similar to SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ, where no actual column values are needed by the query. This array represents an opportunity for optimization. The function need not return all values for all the result columns of the table function. Only a subset of the values may be needed in a particular context, and these are the columns identified (by number) in the array. Since this optimization may complicate the function logic, the function can choose to return every defined column.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a stream file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function can take an action that changes the state of an object that the database manager does not manage. Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action.

NO EXTERNAL ACTION
The function does not perform an external action. It need not be called with each successive function invocation.

NO EXTERNAL ACTION functions might perform better than EXTERNAL ACTION functions because they might not be invoked for each successive function invocation.

FENCED or NOT FENCED
Specifies whether the external function runs in an environment that is isolated from the database manager environment. FENCED is the default.

FENCED
The function will run in a separate thread.

FENCED functions cannot keep SQL cursors open across individual calls to the function. However, the cursors in one thread are independent of the cursors in any other threads which reduces the possibility of cursor name conflicts.

NOT FENCED
The function may run in the same thread as the invoking SQL statement.

NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will

also be preserved between calls to the function. However, cursor names may conflict since the UDF is now running in the same thread as the invoking SQL statement and other NOT FENCED UDFs.

NOT FENCED functions usually perform better than FENCED functions.

PROGRAM TYPE MAIN or PROGRAM TYPE SUB
This parameter is allowed for compatibility with other products. It indicates whether the routine's external program is a program (*PGM) or a procedure in a service program (*SRVPGM).

PROGRAM TYPE MAIN
Specifies that the routine executes as the main entry point in a program. The external program must be a *PGM object.

PROGRAM TYPE SUB
Specifies that the routine executes as a procedure in a service program. The external program must be a *SRVPGM object.

NO FINAL CALL or FINAL CALL
Specifies whether a separate *first call* and *final call* are made to the function. To differentiate between types of calls, the function receives an additional argument that specifies the type of call. For table functions, the *call-type* argument is always present (regardless of whether FINAL CALL or NO FINAL CALL is in effect), and it indicates first call, open call, fetch call, close call, or final call.

With NO FINAL CALL, the database manager will only make three types of calls to the table function: open, fetch and close. However, if FINAL CALL is specified, then in addition to open, fetch and close, a first call and a final call can be made to the table function.

A final call enables the function to free any system resources that it has acquired. A final call is useful when the function has been defined with the SCRATCHPAD keyword and the function acquires system resources and stores them in the scratchpad. The default is NO FINAL CALL.

NO FINAL CALL
Specifies that separate first and final calls are not made to the function. However, the open, fetch, and close calls are still made to the function, and the table function always receives an additional argument that specifies the type of call.

FINAL CALL
Specifies that separate first and final calls are made to the function. It also controls when the scratchpad is re-initialized.

The types of calls are:

First Call
Specifies the first call to the function for this reference to the function in this SQL statement.

Open Call
Specifies a call to open the table function result in this SQL statement.

Fetch Call
Specifies a call to fetch a row from the table function in this SQL statement.

Close Call
Specifies a call to close the table function result in this SQL statement.

Final Call
Specifies the last call to the function to enable the function to free resources. If an error occurs, the database manager attempts to make the final call.

A final call occurs at these times:

- *End of statement:* When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
- *End of transaction:* When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made when the cursor is closed or the application ends.

Committable operations should not be performed during a FINAL CALL, because the FINAL CALL may occur during a close invoked as part of a COMMIT operation.

ALLOW PARALLEL or DISALLOW PARALLEL
Specifies whether the function can be run in parallel.

The default is DISALLOW PARALLEL if one or more of the following clauses are specified: NOT DETERMINISTIC, EXTERNAL ACTION, FINAL CALL, MODIFIES SQL DATA, or SCRATCHPAD. Otherwise, ALLOW PARALLEL is the default.

ALLOW PARALLEL
Specifies that the database manager can consider parallelism for the function. The database manager is not required to use parallelism on the SQL statement that invokes the function or on any SQL statement issued from within the function.

See the descriptions of NOT DETERMINISTIC, EXTERNAL ACTION, MODIFIES SQL DATA, SCRATCHPAD, and FINAL CALL for considerations that apply to specification of ALLOW PARALLEL.

DISALLOW PARALLEL
Specifies that the database manager must not use parallelism for the function.

NO SCRATCHPAD or SCRATCHPAD
Specifies whether the function requires a static memory area.

NO SCRATCHPAD
Specifies that the function does not require a persistent memory area.

SCRATCHPAD *integer*
Specifies that the function requires a persistent memory area of length integer. The integer can range from 1 to 16,000,000. If the memory area is not specified, the size of the area is 100 bytes. If parameter style SQL is specified, a pointer is passed following the required parameters that points to a static storage area. Only 1 memory area will be allocated for the function.

The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad. For example,

assuming that function UDFX was defined with the SCRATCHPAD keyword, two scratchpads are allocated for the two references to UDFX in the following SQL statement:

```
SELECT A.C1, B.C1
  FROM TABLE(UDFX(:hv1)) AS A, TABLE(UDFX(:hv1)) AS B
```

EXTERNAL
Specifies that the CREATE FUNCTION statement is being used to define a new function that is based on code that is written in an external programming language.

If *external-program-name* is not specified, the external program name is assumed to be the same as the function name.

NAME *external-program-name*
Specifies the program, service program, or Java class that will be executed when the function is invoked in an SQL statement. The name must identify a program, service program, or Java class that exists at the application server at the time the function is invoked. If the naming option is *SYS and the name is not qualified:

- The current path will be used to search for the program at the time the function is invoked.
- *LIBL will be used to search for the program or service program at the time COMMENT, GRANT, LABEL, or REVOKE operations are performed on the function.

The validity of the name is checked at the application server. If the format of the name is not correct, an error is returned.

The program, service program, or Java class need not exist at the time the function is created, but it must exist at the time the function is invoked.

CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK, and SET TRANSACTION statements are not allowed in the external program of the function.

CARDINALITY *bigint*
Specifies an estimate of the expected number of rows to be returned by the function for the database manager to use during optimization. *bigint* must be in the range from 0 to 9 223 372 036 854 775 807 inclusive. The database manager assumes a finite value if CARDINALITY is not specified.

A table function that returns a row every time it is called and never returns the end-of-table condition has infinite cardinality. A query that invokes such a function and requires an eventual end-of-table condition before it can return any data will not return unless interrupted. Table functions that never return the end-of-table condition should not be used in queries involving DISTINCT, GROUP BY, or ORDER BY.

Notes

General considerations for defining user-defined functions: See "CREATE FUNCTION" on page 851 for general information about defining user-defined functions.

REPLACE rules: When an external function is recreated by REPLACE:

- Any existing comment or label is discarded.
- If a different external program is specified:
  - Authorized users are not copied to the new program.

  - Journal auditing is not changed.
- Otherwise:
  - Authorized users are maintained. The object owner will not change.
  - Current journal auditing is not changed.

Creating the function: When an external function associated with an ILE external program or service program is created, an attempt is made to save the function's attributes in the associated program or service program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the attributes are used to update the catalogs.

The attributes can be saved for external functions subject to the following restrictions:

- The external program library must not be SYSIBM, QSYS, or QSYS2.
- The external program must exist when the CREATE FUNCTION statement is issued.

  If system naming is specified and the external program name is not qualified, the external program must be found in the library list.
- The external program must be an ILE *PGM or *SRVPGM object.
- The external program must not already contain attributes for 32 routines.

If the object cannot be updated, the function will still be created.

Invoking the function: When an external function is invoked, it runs in whatever activation group was specified when the external program or service program was created. However, ACTGRP(*CALLER) should normally be used so that the function runs in the same activation group as the calling program. ACTGRP(*NEW) is not allowed.

LANGUAGE JAVA functions always run in the default activation group (*DFTACTGRP). Caution should be used when writing MODIFIES SQL DATA Java functions. Since changes performed by the Java function are performed in the default activation group, transaction problems may occur if the invoker runs in a new activation group (*NEW).

Notes for Java functions: To be able to run Java functions, you must have the IBM Developer Kit for Java (5761-JV1) installed on your system. Otherwise, an SQLCODE of -443 will be returned and a CPDB521 message will be placed in the job log.

If an error occurs while running a Java function, an SQLCODE of -443 will be returned. Depending on the error, other messages may exist in the job log of the job where the function was run.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The value DB2GENRL may be used as a synonym for DB2GENERAL.

- The keywords PARAMETER STYLE in the PARAMETER STYLE clause are optional.
- The keywords IS DETERMINISTIC may be used as a synonym for DETERMINISTIC.
- The keywords PARAMETER STYLE DB2SQL may be used as a synonym for PARAMETER STYLE SQL.

Example

The following creates a table function written to return a row consisting of a single document identifier column for each known document in a text management system. The first parameter matches a given subject area and the second parameter contains a given string.

Within the context of a single session, the UDF will always return the same table, and therefore it is defined as DETERMINISTIC. Note the RETURNS clause which defines the output from DOCMATCH. FINAL CALL must be specified for each table function. Although the size of the output for DOCMATCH is highly variable, CARDINALITY 20 is a representative value, and is specified to help the optimizer.

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOCID CHAR(16))
  EXTERNAL NAME 'MYLIB/RAJIV(UDFMATCH)'
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  NOT FENCED
  SCRATCHPAD
  FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20
```

CREATE FUNCTION (Sourced)

This CREATE FUNCTION (Sourced) statement defines a user-defined function, based on another existing scalar or aggregate function, at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization id of the statement must include at least one of the following:

- For the SYSFUNCS catalog view and SYSPARMS catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

If the source function is a user-defined function, the authorization ID of the statement must include at least one of the following for the source function:

- The EXECUTE privilege on the function
- Administrative authority

To create a sourced function, the privileges held by the authorization ID of the statement must also include at least one of the following:

- The following system authorities:
  - *USE to the Create Service Program (CRTSRVPGM) command or
  - *USE to the Create Program (CRTPGM) command
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the function is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

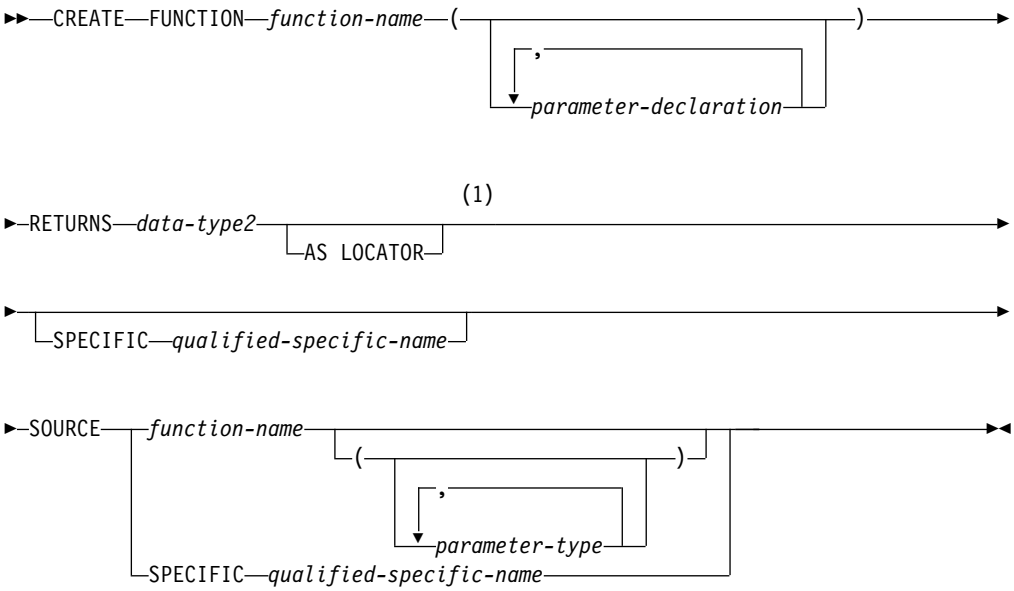
- The system authority *ADD to the user profile with that name
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

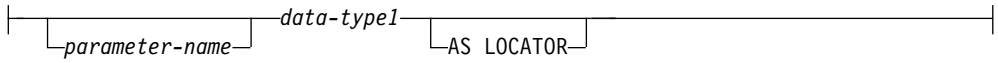
- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View, Corresponding System Authorities When Checking Privileges to a Function or Procedure, and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax



parameter-declaration:

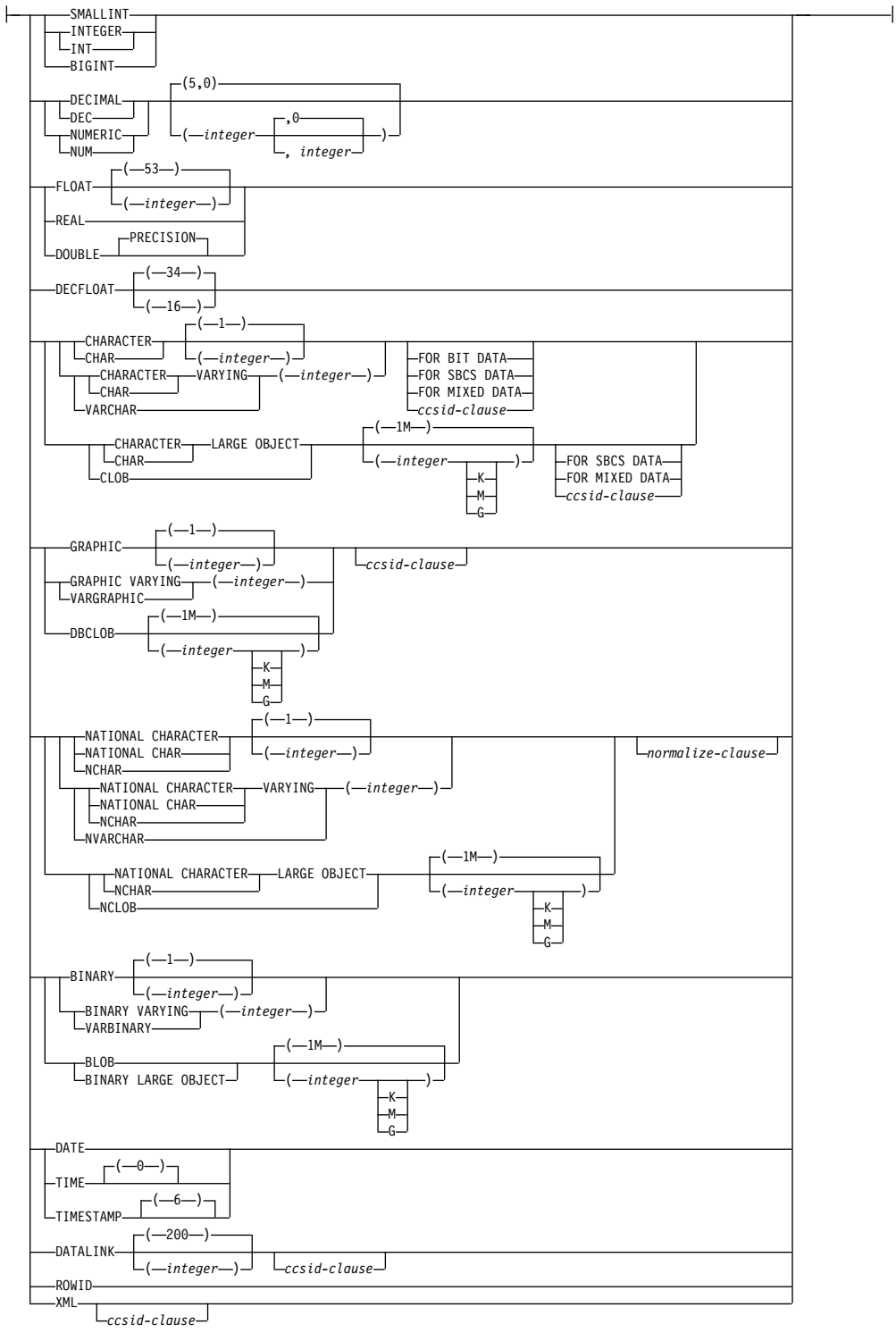


data-type1, data-type2, data-type3:

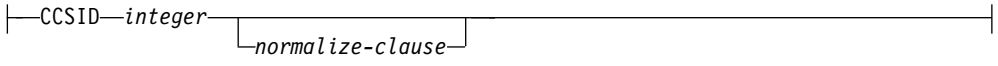
Notes:

1 The RETURNS, SPECIFIC, and SOURCE clauses can be specified in any order.

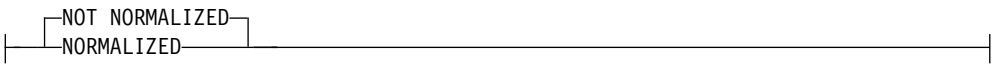
built-in-type:



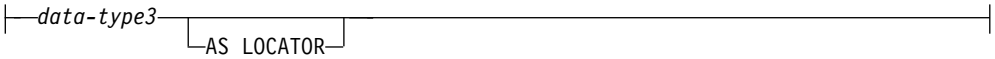
ccsid-clause:



normalize-clause:



parameter-type:



Description

function-name

Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in schema that is specified by the qualifier. If no qualifier is specified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the function will be created in the current library (*CURLIB).
- Otherwise, the function will be created in the current schema.

If the function is sourced on an existing function to enable the use of the existing function with a distinct type, the name can be the same name as the existing function. In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see Choosing the Schema and Function Name.

(*parameter-declaration***,...)**

Specifies the number of input parameters of the function and the data type of each parameter. Each *parameter-declaration* specifies an input parameter for the function. A maximum of 90 parameters can be specified. A function can have zero or more input parameters. There must be one entry in the list for each parameter that the function expects to receive. All the parameters for a function are input parameters and are nullable. In the case of JAVA, numeric parameters other than the DECIMAL and NUMERIC types are not nullable. A runtime error will occur if a null value is input to such a parameter for a CALLED ON NULL INPUT function. For more information, see Defining the parameters.

parameter-name

Names the parameter. Although not required, a parameter name can be specified for each parameter. The name cannot be the same as any other *parameter-name* in the parameter list.

data-type1

Specifies the data type of the parameter. The data type can be a built-in data type or a distinct data type.

Any valid SQL data type may be used, provided that it is castable to the type of the corresponding parameter of the function identified in the

SOURCE clause (for information, see "Casting between data types" on page 95). However, this checking does not guarantee that an error will not occur when the function is invoked. For more information, see Considerations for invoking a sourced user-defined function.

built-in-type
The data type of the input parameter is a built-in data type. See "CREATE TABLE" on page 982 for a more complete description of each built-in data type.

distinct-type-name
The data type of the input parameter is a distinct type. The length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). See "CREATE TYPE (Distinct)" on page 1055 for more information.

If the name of the distinct type is specified without a schema name, the database manager resolves the schema name by searching the schemas in the SQL path.

DataLinks are not allowed for functions sourced on external functions.

If a CCSID is specified, the parameter is converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

AS LOCATOR
Specifies that the input parameter is a locator to the value rather than the actual value. You can specify AS LOCATOR only if the input parameter has a LOB data type or a distinct type based on a LOB data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

RETURNS
Specifies the result of the function.

data-type2
Specifies the data type of the column. The column is nullable. The data type can be a built-in data type (except LONG VARCHAR, LONG VARGRAPHIC, or a DataLink) or distinct type (that is not based on a DataLink).

Any valid SQL data type can be used provided it is castable from the result type of the source function. (For information about casting data types, see "Casting between data types" on page 95) However, this checking does not guarantee that an error will not occur when this new function is invoked. For more information, see Considerations for invoking a sourced user-defined function.

AS LOCATOR
Specifies that the function returns a locator to the value rather than the actual value. You can specify AS LOCATOR only if the output from the function has a LOB data type or a distinct type based on a LOB data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified. The AS LOCATOR clause is not allowed for functions sourced on SQL functions.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

SPECIFIC *qualified-specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a function.

SOURCE
Specifies that the new function is being defined as a sourced function. A *sourced function* is implemented by another function (the *source function*). The function must be a scalar or aggregate function that exists at the current server, and it must be one of the following types of functions:

- A function that was defined with a CREATE FUNCTION statement
- A cast function that was generated by a CREATE TYPE statement
- A built-in function

If the source function is not a built-in function, the particular function can be identified by its name, function signature, or specific name.

If the source function is a built-in function, the SOURCE clause must include a function signature for the built-in function. The source function must not be any of the following built-in functions (If a particular syntax is shown, only the indicated form cannot be specified.):

- ARRAY_AGG
- BINARY when more than one argument is specified
- BLOB when more than one argument is specified
- CARDINALITY
- CHAR when more than one argument is specified
- CLOB when more than one argument is specified
- COALESCE
- CONTAINS
- DATAPARTITIONNAME
- DATAPARTITIONNUM
- DBCLOB when more than one argument is specified
- DBPARTITIONNAME
- DBPARTITIONNUM
- DECFLOAT when more than one argument is specified
- DECIMAL when more than one argument is specified
- DECRYPT_BIN when more than one argument is specified
- DECRYPT_BINARY when more than one argument is specified
- DECRYPT_BIT when more than one argument is specified
- DECRYPT_CHAR when more than one argument is specified
- DECRYPT_DB when more than one argument is specified
- EXTRACT
- GRAPHIC when more than one argument is specified
- HASH
- HASHED_VALUE
- LAND
- LOR
- MAX

- MAX_CARDINALITY
- MIN
- NODENAME
- NODENUMBER
- PARTITION
- POSITION
- RAISE_ERROR
- RID
- RRN
- SCORE
- STRIP
- SUBSTRING
- TRANSLATE when more than one argument is specified
- TRIM
- TRIM_ARRAY
- VALUE
- VARBINARY when more than one argument is specified
- VARCHAR when more than one argument is specified
- VARGRAPHIC when more than one argument is specified
- XMLAGG
- XMLATTRIBUTES
- XMLCOMMENT
- XMLCONCAT
- XMLDOCUMENT
- XMLELEMENT
- XMLFOREST
- XMLGROUP
- XMLNAMESPACES
- XMLPARSE
- XMLPI
- XMLROW
- XMLSERIALIZE
- XMLTEXT
- XMLVALIDATE
- XOR
- XSLTRANSFORM
- ZONED when more than one argument is specified

function-name
Identifies the function to be used as the source function by its function name. The source function can be defined with any number of parameters. If more than one function is defined with the specified name in the specified or implicit schema, an error is returned.

If an unqualified *function-name* is specified, the SQL path is used to locate the function. The database manager selects the first schema that has only one function with this name on which the user has EXECUTE authority.

An error is returned if a function is not found, or if the database manager encounters a schema that has more than one function with this name.

function-name (parameter-type, ...)
Identifies the function to be used as the source function by its function signature, which uniquely identifies the function. The *function-name (parameter-type,...)* must identify a function with the specified signature at the current server. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance. Synonyms for data types are considered a match.

If *function-name*() is specified, the function identified must have zero parameters.

To use a built-in function as the source function, this syntax variation must be used.

function-name
Identifies the name of the source function. If an unqualified name is specified, the schemas of the SQL path are searched. Otherwise, the database manager searches for the function in the specified schema.

parameter-type,...
Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

Empty parentheses are allowed for some data types that are specified in this context. For data types that have a length, precision or scale attribute, use one of the following specifications:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or CCSID attribute, specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB. If AS LOCATOR is specified, FOR

SBCS DATA or FOR MIXED DATA must not be specified. If AS LOCATOR is specified and a length is explicitly specified, the data type length is ignored.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

SPECIFIC *qualified-specific-name*
Identifies the function to be used as the source function by its specific name. The *qualified-specific-name* must identify a specific function that exists in the specified schema.

The number of input parameters in the function that is being created must be the same as the number of parameters in the source function. If the data type of each input parameter is not the same as or castable to the corresponding parameter of the source function, an error occurs. The data type of the final result of the source function must match or be castable to the result of the sourced function.

If a CCSID is specified and the CCSID of the return data is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified the return data is converted to the CCSID of the job (or associated graphic CCSID of the job for graphic string return values), if the CCSID of the return data is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

Notes

General considerations for defining user-defined functions: See "CREATE FUNCTION" on page 851 for general information about defining user-defined functions.

Function ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the function is created exists, the *owner* of the function is that user profile.
- Otherwise, the *owner* of the function is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the function is the user profile or group user profile of the job executing the statement.

Function authority: If SQL names are used, functions are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, functions are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the function is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the function.

Considerations for invoking a sourced user-defined function: When a sourced function is invoked, each argument to the function is assigned to the associated parameter defined for the function. The values are then cast (if necessary) to the

data type of the corresponding parameters of the underlying function. An error can occur either in the assignment or in the cast. For example: an argument passed on input to a function that matches the data type and length or precision attributes of the parameter for the function might not be castable if the corresponding parameter of the underlying source function has a shorter length or less precision. It is recommended that the data types of the parameters of a sourced function be defined with attributes that are less than or equal to the attributes of the corresponding parameters of the underlying function.

The result of the underlying function is assigned to the RETURNS data type of the sourced function. The RETURNS data type of the underlying function might not be castable to the RETURNS data type of the source function. This can occur when the RETURNS data type of this new source function has a shorter length or less precision than the RETURNS data type of the underlying function. For example, an error would occur when function A is invoked assuming the following functions exist. Function A returns an INTEGER. Function B is a sourced function, is defined to return a SMALLINT, and the definition references function A in the SOURCE clause. It is recommended that the RETURNS data type of a sourced function be defined with attributes that are the same or greater than the attributes defined for the RETURNS data type of the underlying function.

Considerations when the function is based on a user-defined function: If the sourced function is based directly or indirectly on an external scalar function, the sourced function inherits the attributes of the external scalar function. This can involve several layers of sourced functions. For example, assume that function A is sourced on function B, which in turn is sourced on function C. Function C is an external scalar function. Functions A and B inherit all of the attributes for function C.

Creating the function: When a sourced function is created, a small service program object is created that represents the function. When this service program is saved and restored to another system, the attributes from the CREATE FUNCTION statement are automatically added to the catalog on that system.

Examples

Example 1: Assume that distinct type HATSIZE is defined and is based on the built-in data type INTEGER. An AVG function could be defined to compute the average hat size of different departments. Create a sourced function that is based on built-in function AVG.

```
CREATE FUNCTION AVG (HATSIZE)
  RETURNS HATSIZE
  SOURCE AVG (INTEGER)
```

The syntax of the SOURCE clause includes an explicit parameter list because the source function is a built-in function.

When distinct type HATSIZE was created, two cast functions were generated, which allow HATSIZE to be cast to INTEGER for the argument and INTEGER to be cast to HATSIZE for the result of the function.

Example 2: After Smith created the external scalar function CENTER in his schema, there is a need to use this function, function, but the invocation of the function needs to accept two INTEGER arguments instead of one INTEGER argument and one DOUBLE argument. Create a sourced function that is based on CENTER.

```
CREATE FUNCTION MYCENTER (INTEGER, INTEGER)
   RETURNS DOUBLE
   SOURCE SMITH.CENTER (INTEGER, DOUBLE);
```

CREATE FUNCTION (SQL Scalar)

This CREATE FUNCTION (SQL Scalar) statement creates an SQL function at the current server. The function returns a single result.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization id of the statement must include at least one of the following:

- For the SYSFUNCS catalog view and SYSPARMS catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative Authority

The privileges held by the authorization ID of the statement must also include at least one of the following:

- The following system authorities:
  - *USE to the Create Service Program (CRTSRVPGM) command or
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

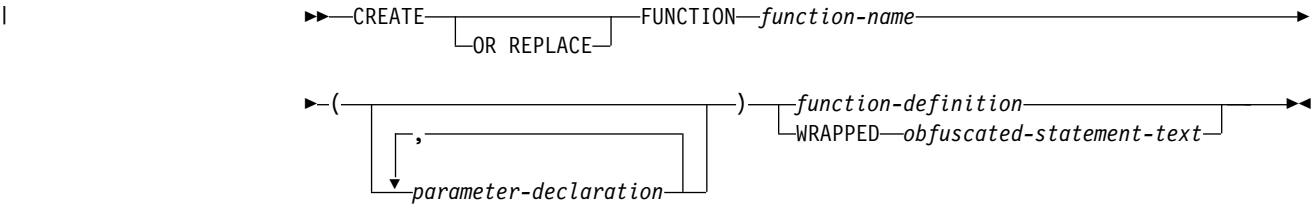
To replace an existing function, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority of *OBJMGT on the service program object associated with the function
  - All authorities needed to DROP the function
  - The system authority *READ to the SYSFUNCS catalog view and SYSPARMS catalog table
- Administrative authority

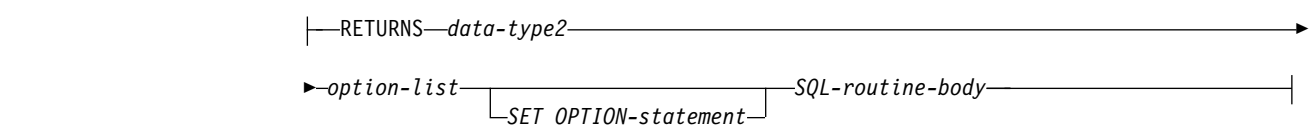
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View

and Corresponding System Authorities When Checking Privileges to a Distinct Type.

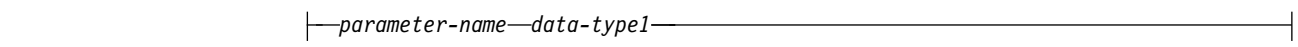
Syntax



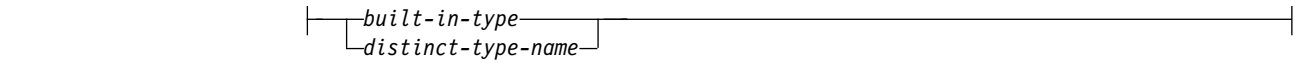
function-definition:



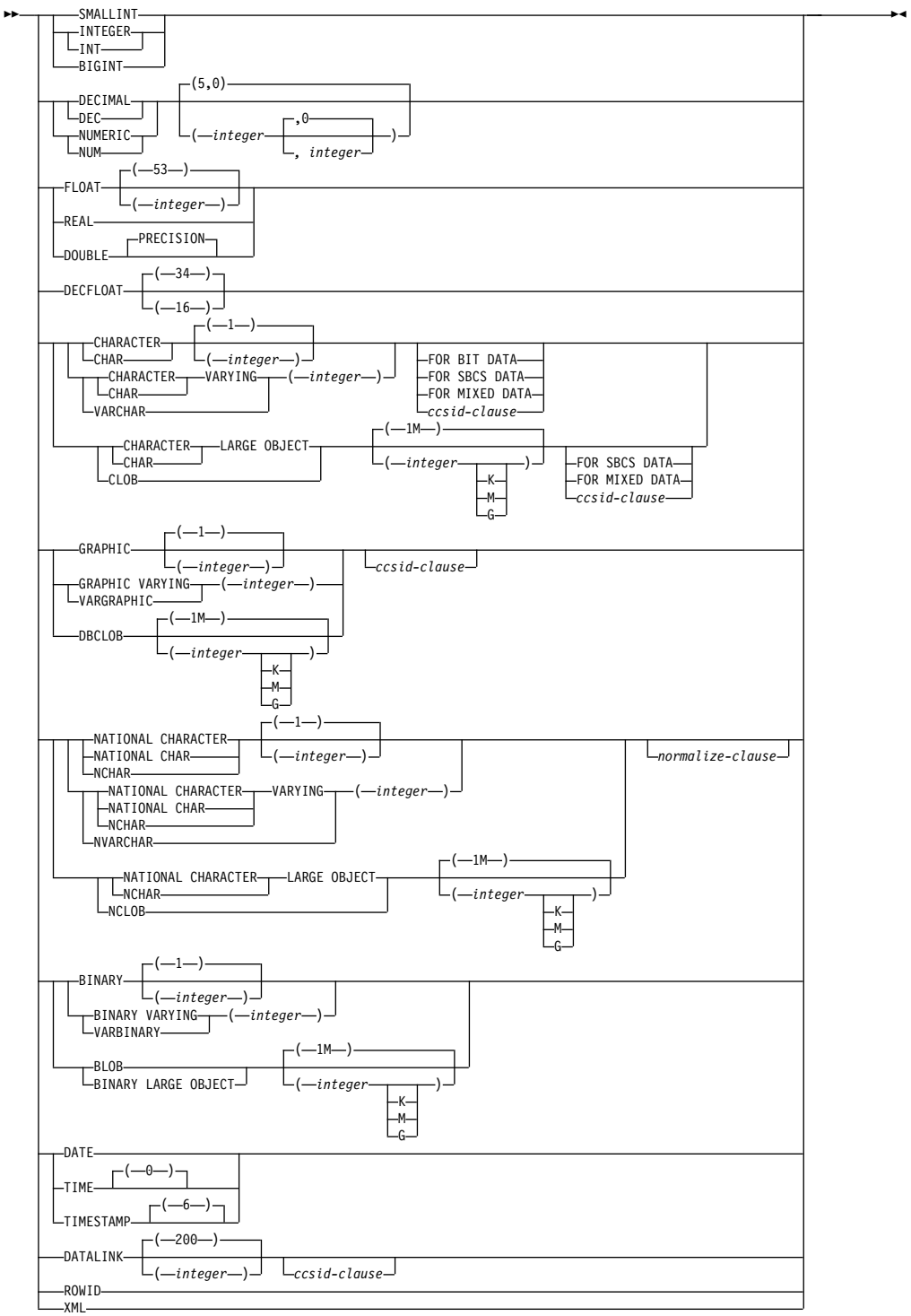
parameter-declaration:



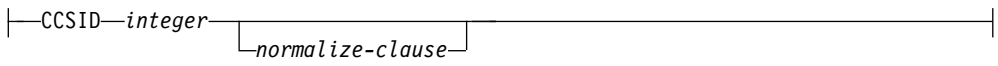
data-type:



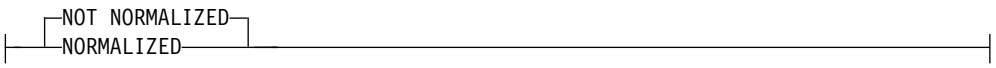
built-in-type



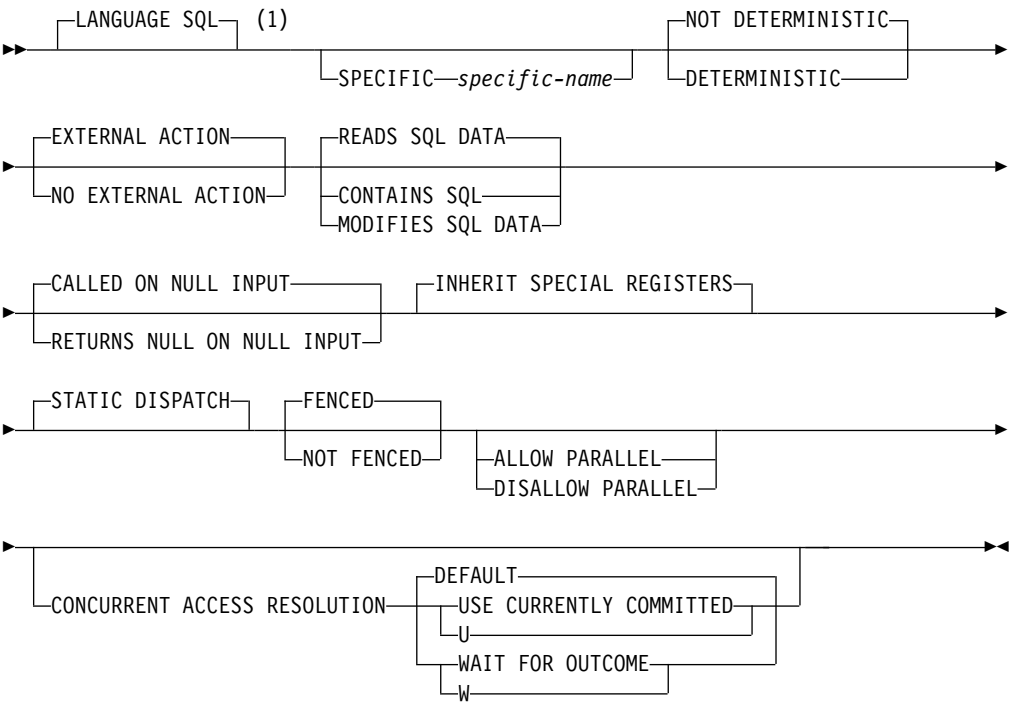
ccsid-clause:



normalize-clause:



option-list



Notes:

1 This clause and the clauses that follow in the option-list can be specified in any order. Each clause can be specified at most once.

SQL-routine-body

Description

OR REPLACE
Specifies to replace the definition for the function if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog with the exception that privileges that were granted on the function are not affected. This option is ignored if a definition for the function does not exist at the current server. To replace an existing function, the *specific-name* and *function-name* of the new definition must be the same as the *specific-name* and *function-name* of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new function is created.

function-name
Names the user-defined function. The combination of name, schema name, the

number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in the schema that is specified by the qualifier. If no qualifier is specified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the function will be created in the current library (*CURLIB).
- Otherwise, the function will be created in the current schema.

In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see Choosing the Schema and Function Name.

(*parameter-declaration,...)*
Specifies the number of input parameters of the function and the data type of each parameter. Each *parameter-declaration* specifies an input parameter for the function. A maximum of 1024 parameters can be specified. A function can have zero or more input parameters. There must be one entry in the list for each parameter that the function expects to receive. All the parameters for a function are input parameters and are nullable. For more information, see Defining the parameters.

parameter-name
Names the parameter. The name is used to refer to the parameter within the body of the function. The name cannot be the same as any other *parameter-name* in the parameter list.

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct data type.

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see "CREATE TABLE" on page 982.

distinct-type-name
Specifies a distinct type. The length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). For more information about creating a distinct type, see "CREATE TYPE (Distinct)" on page 1055.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

If a CCSID is specified, the parameter is converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

RETURNS
Specifies the result of the function.

data-type2
Specifies the expression that is to be returned for the function. The result data type of the expression must be assignable (using storage assignment

rules) to the data type that is defined in the RETURNS clause. For more information, see "Assignments and comparisons" on page 98.

You can specify any built-in data type (except LONG VARCHAR, or LONG VARGRAPHIC) or a distinct type.

If a CCSID is specified and the CCSID of the return data is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified and the function is not referenced in the outermost select list of a view, the return data is converted to the CCSID of the job (or associated graphic CCSID of the job for graphic string return values), if the CCSID of the return data is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

If a CCSID is not specified and the function is referenced in the outermost select list of a view, the return data is converted to the CCSID of the associated view column. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

LANGUAGE SQL
Specifies that this is an SQL function.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a function.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. The database manager uses this information during optimization of SQL statements. An example of a function that is not deterministic is one that generates random numbers.

A function that is not deterministic might return incorrect results if the function is executed by parallel tasks. Specify the DISALLOW PARALLEL clause for these functions.

NOT DETERMINISTIC should be specified if the function contains a reference to a special register, a non-deterministic function, or a sequence.

DETERMINISTIC
Specifies that the function always returns the same result each time that the function is invoked with the same input arguments. The database

manager uses this information during optimization of SQL statements.[91] An example of a deterministic function is a function that calculates the square root of the input argument.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a stream file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function can take an action that changes the state of an object that the database manager does not manage. Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action.

NO EXTERNAL ACTION
The function does not perform an external action. It need not be called with each successive function invocation.

NO EXTERNAL ACTION functions might perform better than EXTERNAL ACTION functions because they might not be invoked for each successive function invocation.

CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA
Specifies the classification of SQL statements that the function can execute. The database manager verifies that the SQL statements that the function issues are consistent with this specification. For the classification of each statement, see Appendix B, "Characteristics of SQL statements," on page 1501. The default is READS SQL DATA.

READS SQL DATA
Specifies that the function can execute statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access classification of CONTAINS SQL or NO SQL. The function cannot execute any SQL statements that read or modify data.

MODIFIES SQL DATA
The function can execute any SQL statement except those statements that are not supported in any function.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON INPUT
Specifies that the function is not invoked if any of the input arguments is null. The result is the null value.

CALLED ON NULL INPUT
Specifies that the function is to be invoked if any or all argument values

91. The query optimizer may choose to cache deterministic scalar function results. The DETERMINISTIC_UDF_SCOPE QAQQINI option can be used to control the scope of the caching. If the result of the function contains sensitive data, consider using the DETERMINISTIC_UDF_SCOPE QAQQINI option or create the function NOT DETERMINISTIC to prevent inadvertent access to the result. For more information, see the Database Performance and Query Optimization topic collection.

are null. This specification means that the function must be coded to test for null argument values. The function can return a null or nonnull value.

INHERIT SPECIAL REGISTERS
Specifies that existing values of special registers are inherited upon entry to the function.

STATIC DISPATCH
Specifies that the function is dispatched statically. All functions are statically dispatched.

FENCED or NOT FENCED
Specifies whether the SQL function runs in an environment that is isolated from the database manager environment. FENCED is the default.

FENCED
The function will run in a separate thread.

FENCED functions cannot keep SQL cursors open across individual calls to the function. However, the cursors in one thread are independent of the cursors in any other threads which reduces the possibility of cursor name conflicts.

NOT FENCED
The function may run in the same thread as the invoking SQL statement.

NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will also be preserved between calls to the function. However, cursor names may conflict since the UDF is now running in the same thread as the invoking SQL statement and other NOT FENCED UDFs.

NOT FENCED functions usually perform better than FENCED functions.

ALLOW PARALLEL or DISALLOW PARALLEL
Specifies whether the function can be run in parallel.

The default is DISALLOW PARALLEL if one or more of the following clauses are specified: NOT DETERMINISTIC, EXTERNAL ACTION, or MODIFIES SQL DATA. Otherwise, ALLOW PARALLEL is the default.

ALLOW PARALLEL
Specifies that the database manager can consider parallelism for the function. The database manager is not required to use parallelism on the SQL statement that invokes the function or on any SQL statement issued from within the function.

See the descriptions of NOT DETERMINISTIC, EXTERNAL ACTION, and MODIFIES SQL DATA for considerations that apply to specification of ALLOW PARALLEL.

DISALLOW PARALLEL
Specifies that the database manager must not use parallelism for the function.

CONCURRENT ACCESS RESOLUTION
Specifies whether the database manager should wait for data that is in the process of being updated. DEFAULT is the default.

DEFAULT
Specifies that the concurrent access resolution is not explicitly set for this function. The value that is in effect when the function is invoked will be used.

WAIT FOR OUTCOME
Specifies that the database manager is to wait for the commit or rollback of data in the process of being updated.

USE CURRENTLY COMMITTED
Specifies that the database manager is to use the currently committed version of the data when encountering data that is in the process of being updated.

When the lock contention is between a read transaction and a delete or update transaction, the clause is applicable to scans with isolation level CS (but not for CS KEEP LOCKS).

WRAPPED *obfuscated-statement-text*
Specifies the encoded definition of the function. A CREATE FUNCTION statement can be encoded using the WRAP scalar function.

SET OPTION-statement
Specifies the options that will be used to create the function. For example, to create a debuggable function, the following statement could be included:

```
SET OPTION DBGVIEW = *SOURCE
```

The default values for the options depend the options in effect at create time. For information about the , see "SET OPTION" on page 1368.

The options CNULRQD, CNULIGN, COMPILEOPT, NAMING, and SQLCA are not allowed in the CREATE FUNCTION statement.

SQL-routine-body
Specifies a single *SQL-procedure-statement*, including a compound statement. See Chapter 7, "SQL control statements," on page 1427 for more information about defining SQL functions.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK, and SET TRANSACTION statement is not allowed in a function.

If the *SQL-routine-body* is a compound statement, it must contain at least one RETURN statement and a RETURN statement must be executed when the function is called.

ALTER PROCEDURE (SQL), ALTER FUNCTION (SQL Scalar), and ALTER FUNCTION (SQL Table) with a REPLACE keyword are not allowed in an *SQL-routine-body*.

Notes

General considerations for defining user-defined functions: For general information about defining user-defined functions, see "CREATE FUNCTION" on page 851.

SQL path and function resolution: Resolution of function invocations inside the function body is done according to the SQL path that is in effect for the CREATE FUNCTION statement and does not change after the function is created.

Function ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the function is created exists, the *owner* of the function is that user profile.

- Otherwise, the *owner* of the function is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the function is the user profile or group user profile of the job executing the statement.

Function authority: If SQL names are used, functions are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, functions are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the function is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the function.

REPLACE rules: When a function is recreated by REPLACE:

- Any existing comment or label is discarded.
- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved.

If the function is replaced and the function signature or result data type is altered, the results from any function, materialized query table, procedure, trigger, or view that references the function may be unpredictable. Any referenced objects should be recreated.

Creating the function: When an SQL function is created, the database manager creates a temporary source file that will contain C source code with embedded SQL statements. A *SRVPGM object is then created using the CRTSRVPGM command. The SQL options used to create the service program are the options that are in effect at the time the CREATE FUNCTION statement is executed. The service program is created with ACTGRP(*CALLER).

When an SQL function is created, the function's attributes are stored in the created service program object. If the *SRVPGM object is saved and then restored to this or another system, the attributes are used to update the catalogs.

The specific name is used to determine the name of the source file member and *SRVPGM object. If the specific name is a valid system name, it will used as the name of member and program. If the member already exists, it will be overlaid. If a program already exists in the specified library, a unique name is generated using the rules for generating system table names. If the specific name is not a valid system name, a unique name is generated using the rules for generating system table names.

Invoking the function: When an SQL function is invoked, it runs in the activation group of the calling program.

If a function is specified in the *select-list* of a *select-statement* and if the function specifies EXTERNAL ACTION or MODIFIES SQL DATA, the function will only be invoked for each row returned. Otherwise, the UDF may be invoked for rows that are not selected.

Inline functions: In cases of very simple SQL functions, instead of invoking the function as part of a query, the *expression* in the RETURN statement of the function may be copied (inlined) into the query itself. Such a function is called an *inline function*. A function is an *inline function* if:

- The SQL function is deterministic
- The *SQL-routine-body* contains only a RETURN statement.
- The RETURN statement does not contain a scalar subselect or fullselect.
- The *SQL-routine-body* does not contain BEGIN ATOMIC.

An *inline function* is only copied (inlined) into a query if:

- The query must be eligible for the SQL Query Engine (SQE).
- If the function references an object, the authority attributes of the function and the query must be compatible:
  - If the function is defined to run under the user's authority (*USER), the function can be inlined.
  - If both the function and query run under the owner's authority (*OWNER) and the owner for both is the same, the function can be inlined.
  - Otherwise, the function cannot be inlined.
- The following attributes of the function must match the attributes of the query.
  - DATFMT, DATSEP, TIMFMT, and TIMSEP
  - SRTSEQ and LANGID (if SRTSEQ is *LANGIDSHR or *LANGIDUNQ)
  - DECFLTRND (if there is a DECFLOAT field used in the function)
  - DECMPT and DECRESULT
  - SQLPATH (if there is an object reference in the function)
  - The CCSID of constants (the source CCSID) in the function and query

If a function is inlined and it contains a reference to a special register, the value of the special register will be the same as other references to the same special register in the query.

If a function is inlined and it is defined with MODIFIES SQL DATA, the MODIFIES SQL DATA attribute is ignored.

Obfuscated statements: A CREATE FUNCTION statement can be executed in obfuscated form. In an obfuscated statement, only the function name and parameters are readable followed by the WRAPPED keyword. The rest of the statement is encoded in such a way that it is not readable but can be decoded by a database server that supports obfuscated statements. Obfuscated statements can be produced by invoking the WRAP scalar function. Any debug options that are specified when the function is created from an obfuscated statement are ignored. A function that is created from an obfuscated statement cannot be restored to a release where obfuscation is not supported.

Dependent objects: An SQL routine is dependent on objects that are referenced in the *SQL-routine-body*. The names of the dependent objects are stored in catalog view SYSROUTINEDEP. If the object reference in the *SQL-routine-body* is a fully qualified name or, in SQL naming, if an unqualified name is qualified by the current schema, then the schema name of the object in SYSROUTINEDEP will be set to the specified name or the value of the current schema. Otherwise, the schema name is not set to a specific schema name. If a name is not set to a specific schema name, then DROP and ALTER statements will not be able to determine whether the routine is dependent on the object being altered or dropped.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords IS DETERMINISTIC may be used as a synonym for DETERMINISTIC.

Example

Example 1: Define a scalar function that returns the tangent of a value using the existing SIN and COS built-in functions.

```
CREATE FUNCTION TAN
     (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(X)/COS(X)
```

Notice that a parameter name (X) is specified for the input parameter to function TAN. The parameter name is used within the body of the function to refer to the input parameter. The invocations of the SIN and COS functions, within the body of the TAN user-defined function, pass the parameter X as input.

Example 2: Define a scalar function that returns a date formatted as *mm/dd/yyyy* followed by a string of up to 3 characters:

```
CREATE FUNCTION BADPARM
     (INP1 DATE,)
      USA  VARCHAR(3))
  RETURNS VARCHAR(20)
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN CHAR(INP1,USA)CONCAT USA
```

Assume that the function is invoked as in the following statement:

```
SELECT BADPARM(BIRTHDATE,'ISO')
    FROM EMPLOYEE WHERE EMPNO='000010'
```

The result is '08/24/1933ISO'. Notice that parameter names (INP1 and USA) are specified for the input parameters to function BADPARM. Although there is an input parameter named USA, the instance of USA in the parameter list for the CHAR function is taken as the keyword parameter for the built-in CHAR function and not the parameter named USA.

CREATE FUNCTION (SQL Table)

This CREATE FUNCTION (SQL table) statement creates an SQL table function at the current server. The function returns a single result table.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization id of the statement must include at least one of the following:

- For the SYSFUNCS catalog view and SYSPARMS catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative Authority

The privileges held by the authorization ID of the statement must also include at least one of the following:

- The following system authorities:
  - *USE to the Create Service Program (CRTSRVPGM) command or
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

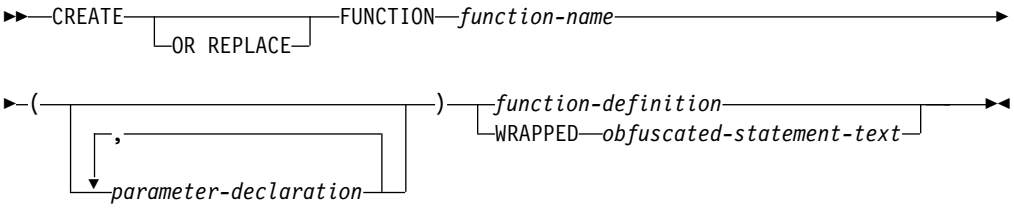
To replace an existing function, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority of *OBJMGT on the service program object associated with the function
  - All authorities needed to DROP the function
  - The system authority *READ to the SYSFUNCS catalog view and SYSPARMS catalog table
- Administrative authority

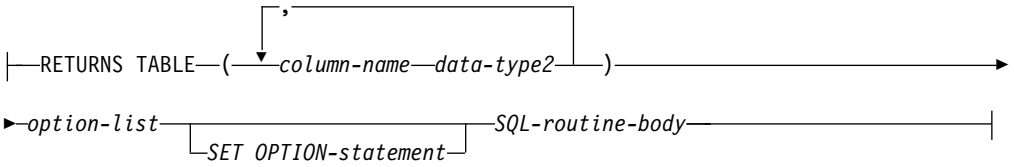
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View

and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax

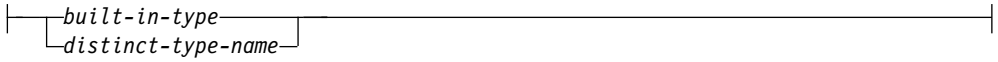


function-definition:

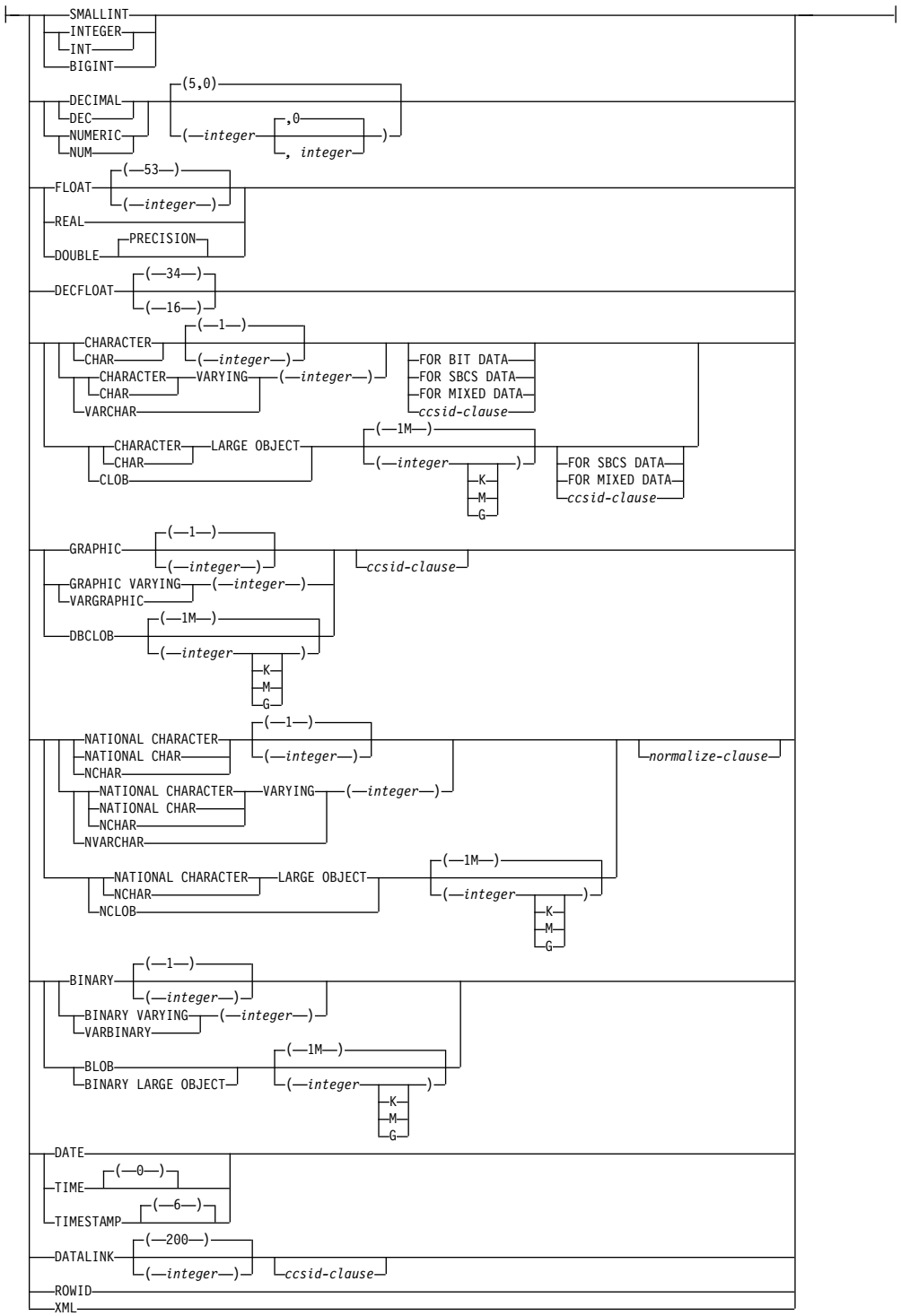


parameter-declaration:

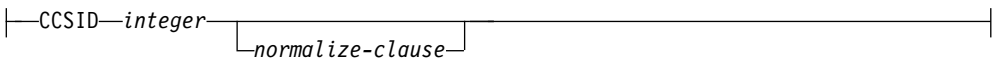
data-type1, data-type2:



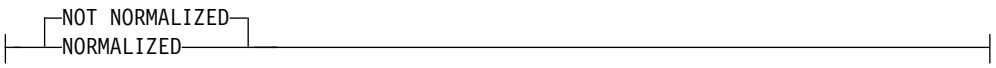
built-in-type:



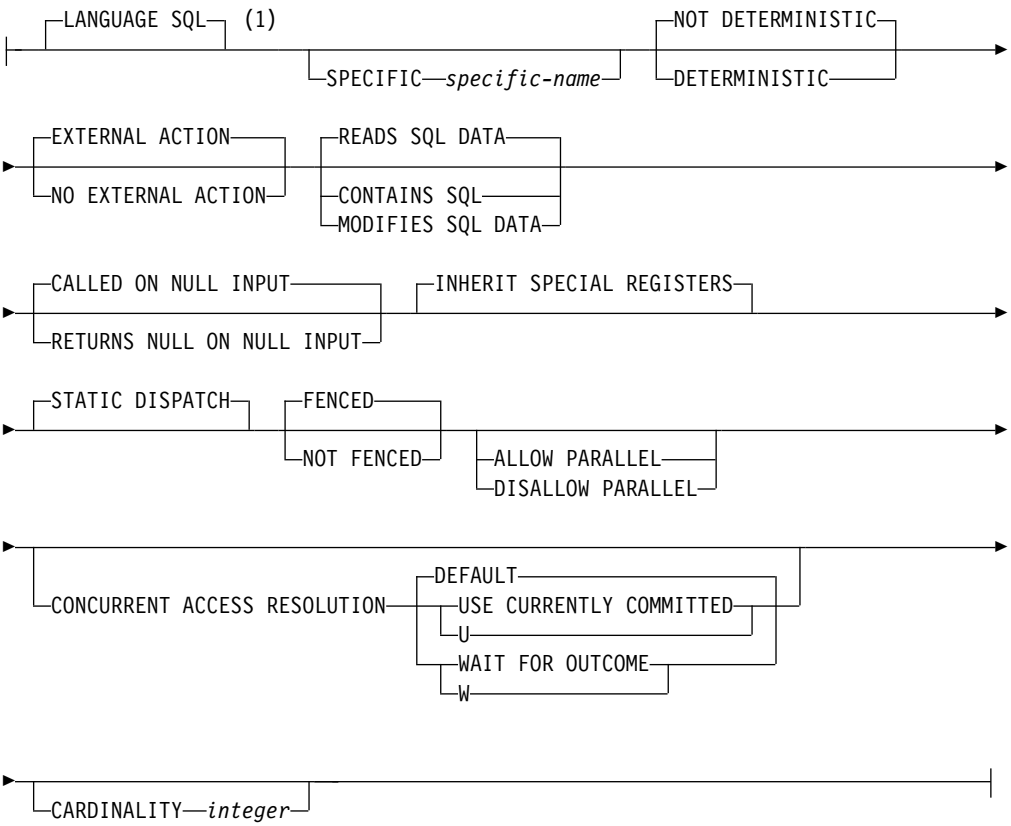
ccsid-clause:



normalize-clause:



option-list:



Notes:

1 This clause and the clauses that follow in the option-list can be specified in any order. Each clause can be specified at most once.

SQL-routine-body:

Description

OR REPLACE
Specifies to replace the definition for the function if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog with the exception that privileges that were granted on the function are not affected. This option is ignored if a definition for the function does not exist at the current server. To replace an existing function, the *specific-name* and *function-name* of the new definition must be the same as

the *specific-name* and *function-name* of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new function is created.

function-name
Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in the schema that is specified by the qualifier. If no qualifier is specified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the function will be created in the current library (*CURLIB).
- Otherwise, the function will be created in the current schema.

In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see Choosing the Schema and Function Name.

(*parameter-declaration,...)*
Specifies the number of input parameters of the function and the data type of each parameter. Each *parameter-declaration* specifies an input parameter for the function. A maximum of 1024 parameters can be specified. A function can have zero or more input parameters. There must be one entry in the list for each parameter that the function expects to receive. All the parameters for a function are input parameters and are nullable. For more information, see Defining the parameters.

parameter-name
Names the parameter. The name is used to refer to the parameter within the body of the function. The name cannot be the same as any other *parameter-name* in the parameter list.

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct data type.

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see “CREATE TABLE” on page 982.

distinct-type-name
Specifies a distinct type. The length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). For more information about creating a distinct type, see “CREATE TYPE (Distinct)” on page 1055.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

RETURNS TABLE
Specifies the output table of the function.

Assume the number of parameters is *N*. There must be no more than 1025-*N* columns.

column-name
Specifies the name of a column of the output table. Do not specify the same name more than once.

data-type2
Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, or LONG VARGRAPHIC) or a distinct type. When the function is invoked the results are assigned to these data types (using storage assignment rules).

If a CCSID is specified and the CCSID of the return data is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified and the function is not referenced in a view, the return data is converted to the CCSID of the job (or associated graphic CCSID of the job for graphic string return values), if the CCSID of the return data is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

If a CCSID is not specified and the function is referenced in a view, the return data is converted to the CCSID of the associated view column. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 1200 or 13488 (Unicode graphic string data).

LANGUAGE SQL
Specifies that this is an SQL function.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a function.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. The database manager uses this information during optimization of SQL statements. An example of a table function that is not deterministic is one that references special registers, non-deterministic functions, or a sequence in a way that affects the table function result table.

DETERMINISTIC
Specifies that the function always returns the same result table each time

that the function is invoked with the same input arguments. The database manager uses this information during optimization of SQL statements.[92]

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a stream file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function can take an action that changes the state of an object that the database manager does not manage. Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action.

NO EXTERNAL ACTION
The function does not perform an external action. It need not be called with each successive function invocation.

NO EXTERNAL ACTION functions might perform better than EXTERNAL ACTION functions because they might not be invoked for each successive function invocation.

CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA
Specifies the classification of SQL statements that the function can execute. The database manager verifies that the SQL statements that the function issues are consistent with this specification. For the classification of each statement, see Appendix B, "Characteristics of SQL statements," on page 1501. The default is READS SQL DATA.

READS SQL DATA
Specifies that the function can execute statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access classification of CONTAINS SQL or NO SQL. The function cannot execute any SQL statements that read or modify data.

MODIFIES SQL DATA
The function can execute any SQL statement except those statements that are not supported in any function.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON NULL INPUT
Specifies that the function is not called if any of the input arguments is null. The result is an empty table, which is a table with no rows. RETURNS NULL ON NULL INPUT is the default.

CALLED ON NULL INPUT
Specifies that the function is to be invoked if any argument values are null.

92. The query optimizer may choose to cache deterministic table function results. The DETERMINISTIC_UDF_SCOPE QAQQINI option can be used to control the scope of the caching. For more information, see the Database Performance and Query Optimization topic collection.

This specification means that the function must be coded to test for null argument values. The function can return an empty table, depending on its logic.

INHERIT SPECIAL REGISTERS
Specifies that existing values of special registers are inherited upon entry to the function.

STATIC DISPATCH
Specifies that the function is dispatched statically. All functions are statically dispatched.

FENCED or NOT FENCED
Specifies whether the SQL function runs in an environment that is isolated from the database manager environment. FENCED is the default.

FENCED
The function will run in a separate thread.

FENCED functions cannot keep SQL cursors open across individual calls to the function. However, the cursors in one thread are independent of the cursors in any other threads which reduces the possibility of cursor name conflicts.

NOT FENCED
The function may run in the same thread as the invoking SQL statement.

NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will also be preserved between calls to the function. However, cursor names may conflict since the UDF is now running in the same thread as the invoking SQL statement and other NOT FENCED UDFs.

NOT FENCED functions usually perform better than FENCED functions.

ALLOW PARALLEL or DISALLOW PARALLEL
Specifies whether the function can be run in parallel.

The default is DISALLOW PARALLEL if one or more of the following clauses are specified: NOT DETERMINISTIC, EXTERNAL ACTION, or MODIFIES SQL DATA, or if this is a pipelined table function. Otherwise, ALLOW PARALLEL is the default.

ALLOW PARALLEL
Specifies that the database manager can consider parallelism for the function. The database manager is not required to use parallelism on the SQL statement that invokes the function or on any SQL statement issued from within the function.

See the descriptions of NOT DETERMINISTIC, EXTERNAL ACTION, and MODIFIES SQL DATA for considerations that apply to specification of ALLOW PARALLEL.

DISALLOW PARALLEL
Specifies that the database manager must not use parallelism for the function.

CONCURRENT ACCESS RESOLUTION
Specifies whether the database manager should wait for data that is in the process of being updated. DEFAULT is the default.

DEFAULT
Specifies that the concurrent access resolution is not explicitly set for this function. The value that is in effect when the function is invoked will be used.

WAIT FOR OUTCOME
Specifies that the database manager is to wait for the commit or rollback of data in the process of being updated.

USE CURRENTLY COMMITTED
Specifies that the database manager is to use the currently committed version of the data when encountering data that is in the process of being updated.

When the lock contention is between a read transaction and a delete or update transaction, the clause is applicable to scans with isolation level CS (but not for CS KEEP LOCKS).

WRAPPED *obfuscated-statement-text*
Specifies the encoded definition of the function. A CREATE FUNCTION statement can be encoded using the WRAP scalar function.

CARDINALITY *integer*
This optional clause provides an estimate of the expected number of rows to be returned by the function for optimization purposes. Valid values for integer range from 0 to 2 147 483 647 inclusive.

If the CARDINALITY clause is not specified for a table function, the database manager will assume a finite value as a default.

A table function that returns a row every time it is called and never returns the end-of-table condition has infinite cardinality. A query that invokes such a function and requires an eventual end-of-table condition before it can return any data will not return unless interrupted.

SET OPTION-statement
Specifies the options that will be used to create the function. For example, to create a debuggable function, the following statement could be included:

```
SET OPTION DBGVIEW = *SOURCE
```

The default values for the options depend the options in effect at create time. For more information, see "SET OPTION" on page 1368.

The options CNULRQD, COMPILEOPT, NAMING, and SQLCA are not allowed in the CREATE FUNCTION statement. CLOSQLCSR(*ENDACTGRP) is always used for SQL table functions.

SQL-routine-body
Specifies a single SQL statement, including a compound statement. See Chapter 7, "SQL control statements," on page 1427 for more information about defining SQL functions.

A non-pipelined table function must contain exactly one RETURN statement. A pipelined table function must contain at least one RETURN statement. The RETURN statement must be executed when the function is invoked.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK and SET TRANSACTION statement is not allowed in a function.

If the *SQL-routine-body* is a compound statement, it must contain exactly one RETURN statement and it must be executed when the function is called.

ALTER PROCEDURE (SQL), ALTER FUNCTION (SQL Scalar), and ALTER FUNCTION (SQL Table) with a REPLACE keyword are not allowed in an *SQL-routine-body*.

Notes

General considerations for defining user-defined functions: See Chapter 7, "SQL control statements," on page 1427 for general information about defining user-defined functions.

Function ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the function is created exists, the *owner* of the function is that user profile.
- Otherwise, the *owner* of the function is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the function is the user profile or group user profile of the job executing the statement.

Function authority: If SQL names are used, functions are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, functions are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the function is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the function.

Pipelined and non-pipelined functions: There are two types of SQL table functions. A table function which does not contain any PIPE statements within the *SQL-routine-body* is a non-pipelined table function. It contains one RETURN statement which returns a table. A pipelined table function is a table function which contains one or more RETURN statements with no return values and zero or more PIPE statements within the *SQL-routine-body*. It returns a table a row at a time. The two types of table functions are invoked in exactly the same way.

A PIPE statement returns a result row from the table function. To get the next row, control returns in the *SQL-routine-body* to the statement following the PIPE statement.

REPLACE rules: When a function is recreated by REPLACE:

- Any existing comment or label is discarded.
- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved.

If the function is replaced and the function signature or result data types are altered, the results from any function, materialized query table, procedure, trigger, or view that references the function may be unpredictable. Any referenced objects should be recreated.

Creating the function: When an SQL function is created, the database manager creates a temporary source file that will contain C source code with embedded SQL statements. A *SRVPGM object is then created using the CRTSRVPGM command. The SQL options used to create the service program are the options that

are in effect at the time the CREATE FUNCTION statement is executed. The service program is created with ACTGRP(*CALLER).

When an SQL function is created, the function's attributes are stored in the created service program object. If the *SRVPGM object is saved and then restored to this or another system, the attributes are used to update the catalogs.

The specific name is used to determine the name of the source file member and *SRVPGM object. If the specific name is a valid system name, it will used as the name of member and program. If the member already exists, it will be overlaid. If a program already exists in the specified library, a unique name is generated using the rules for generating system table names. If the specific name is not a valid system name, a unique name is generated using the rules for generating system table names.

Invoking the function: When an SQL function is invoked, it runs in the activation group of the calling program.

Obfuscated statements: A CREATE FUNCTION statement can be executed in obfuscated form. In an obfuscated statement, only the function name and parameters are readable followed by the WRAPPED keyword. The rest of the statement is encoded in such a way that it is not readable but can be decoded by a database server that supports obfuscated statements. Obfuscated statements can be produced by invoking the WRAP scalar function. Any debug options that are specified when the function is created from an obfuscated statement are ignored. A function that is created from an obfuscated statement cannot be restored to a release where obfuscation is not supported.

Dependent objects: An SQL routine is dependent on objects that are referenced in the *SQL-routine-body*. The names of the dependent objects are stored in catalog view SYSROUTINEDEP. If the object reference in the *SQL-routine-body* is a fully qualified name or, in SQL naming, if an unqualified name is qualified by the current schema, then the schema name of the object in SYSROUTINEDEP will be set to the specified name or the value of the current schema. Otherwise, the schema name is not set to a specific schema name. If a name is not set to a specific schema name, then DROP and ALTER statements will not be able to determine whether the routine is dependent on the object being altered or dropped.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords IS DETERMINISTIC may be used as a synonym for DETERMINISTIC.

Example

Define a table function that returns the employees in a specified department number.

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO CHAR(3))
  RETURNS TABLE (EMPNO CHAR(6),
                 LASTNAME VARCHAR(15),
                 FIRSTNAME VARCHAR(12))
```

```
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
DETERMINISTIC
DISALLOW PARALLEL
RETURN
    SELECT EMPNO,LASTNAME,FIRSTNME
      FROM EMPLOYEE
      WHERE EMPLOYEE.WORKDEPT =DEPTEMPLOYEES.DEPTNO
```

CREATE INDEX

The CREATE INDEX statement creates an index on a table at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - *USE to the Create Logical File (CRTLF) command
  - *CHANGE to the data dictionary if the library into which the index is created is an SQL schema with a data dictionary
- Administrative authority

The privileges held by the authorization ID of the statement must also include at least one of the following:

- For the referenced table:
  - The INDEX privilege on the table
  - The system authority *EXECUTE on the library containing the table
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the table is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

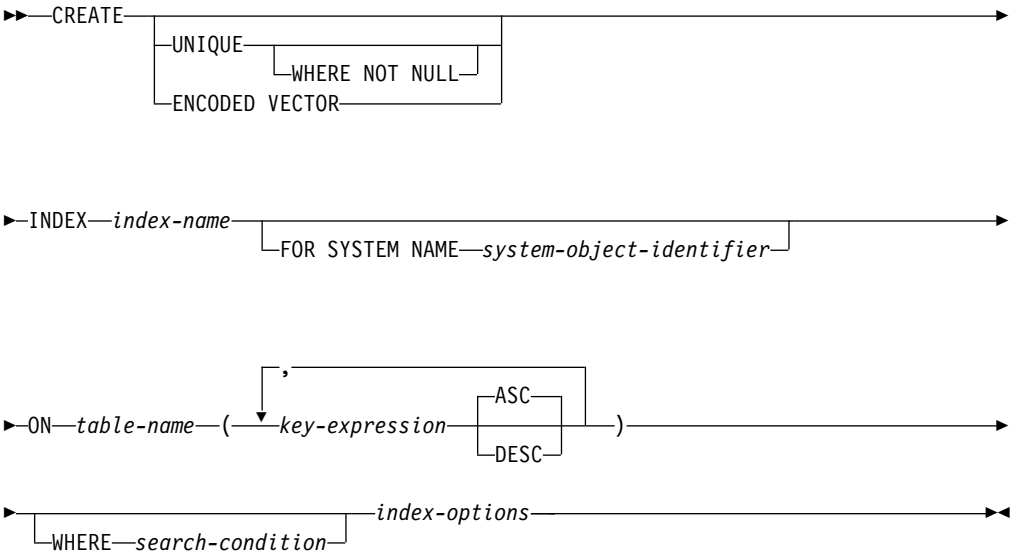
- The system authority *ADD to the user profile with that name
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

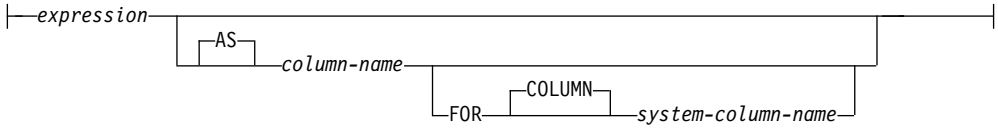
- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

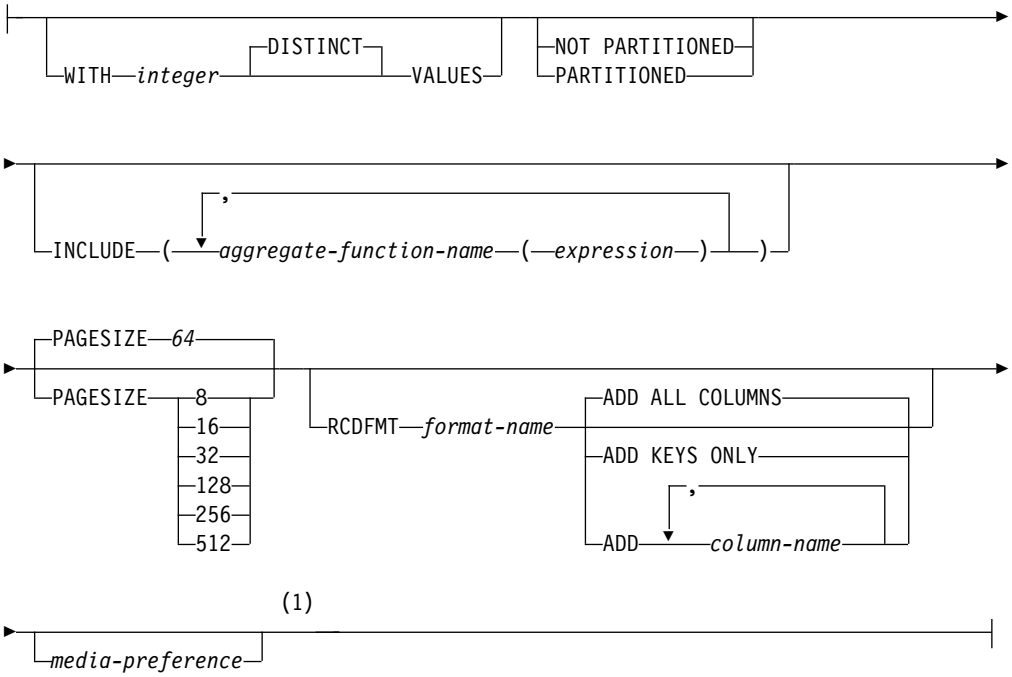
Syntax



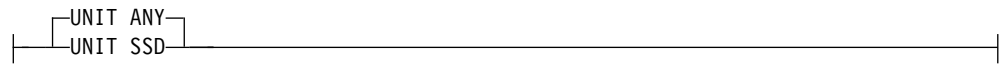
key-expression:



index-options:



media-preference:



Notes:

1 The *index-options* may be specified in any order.

Description

UNIQUE
Prevents the table from containing two or more rows with the same value of the index key. When UNIQUE is used, all null values for a column are considered equal. For example, if the key is a single column that can contain null values, that column can contain only one null value. The constraint is enforced when rows of the table are updated or new rows are inserted.

The constraint is also checked during the execution of the CREATE INDEX statement. If the table already contains rows with duplicate key values, the index is not created.

UNIQUE WHERE NOT NULL
Prevents the table from containing two or more rows with the same value of the index key, where all null values for a column are not considered equal. Multiple null values in a column are allowed. Otherwise, this is identical to UNIQUE.

ENCODED VECTOR
Specifies that the resulting index will be an encoded vector index (EVI).

An encoded vector index cannot be used to ensure an ordering of rows. It is used by the database manager to improve the performance of queries. For more information, see the Database Performance and Query Optimization topic collection.

index-name
Names the index. The name, including the implicit or explicit qualifier, must not be the same as an index, table, view, alias, or file that already exists at the current server.

If SQL names were specified, the index will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the index name will be created in the schema that is specified by the qualifier. If not qualified, the index name will be created in the same schema as the table over which the index is created.

If the index name is not a valid system name and the FOR SYSTEM NAME clause is not used, DB2 for i will generate a system name. For information about the rules for generating a name, see "Rules for Table Name Generation" on page 1030.

FOR SYSTEM NAME *system-object-identifier*
Identifies the *system-object-identifier* of the index. *system-object-identifier* must not be the same as a table, view, alias, or index that already exists at the current server. The *system-object-identifier* must be an unqualified system identifier.

When *system-object-identifier* is specified, *index-name* must not be a valid system object name.

ON *table-name*
Identifies the table on which the index is to be created. The *table-name* must identify a base table (not a view) that exists at the current server.

If the table is a partitioned table, an alias may be specified which identifies a single partition. The created index will then only be created over the specified partition.

key-expression
Identifies a column or expression that will be part of the index key.

The number of keys defined for the index must not exceed 120, and the sum of their byte lengths must not exceed 32766-*n*, where *n* is the number of keys specified that allow nulls.

expression
If *expression* contains only a *column-name*, it must be an unqualified name that identifies a column of the table. The same *column-name* cannot be specified more than once if:

- a WHERE clause, INCLUDE clause, or RCDFMT clause is specified,
- an expression is defined as part of an index key, or
- a column is renamed using the AS clause.

If the *expression* is not a column name, the *expression* must not reference a column that contains a field procedure.

A *column-name* must not identify a LOB, XML, or DATALINK column, or a distinct type based on a LOB, XML, or DATALINK column. If the *expression* is not a column name, any intermediate result expression and the final result expression must not be a DATALINK, LOB, or XML data type. It must not contain any of the following:

- Subqueries
- Aggregate functions
- Variables
- Global variables
- Parameter markers
- Complex expressions that contain LOBs (such as concatenation)
- Special registers
- Sequence references
- OLAP specifications
- REGEXP_LIKE predicate
- ROW CHANGE expressions
- User-defined functions other than functions that were implicitly generated with the creation of a distinct type
- Any function that is not deterministic
- The following built-in scalar functions:

ATAN2	DLURLCOMPLETE [1]	LPAD	REPLACE
CARDINALITY	DLURLPATH	MAX_CARDINALITY	ROUND_TIMESTAMP
CONTAINS	DLURLPATHONLY	MONTHNAME	RPAD
CURDATE	DLURLSCHEME	MONTHS_BETWEEN	SCORE
CURTIME	DLURLSERVER	NEXT_DAY	SOUNDEX
DATAPARTITIONNAME	DLVALUE	NOW	TIMESTAMP_FORMAT
DATAPARTITIONNUM	ENCRYPT_AES	OVERLAY	TIMESTAMPDIFF
DAYNAME	ENCRYPT_RC2	RAISE_ERROR	TRUNC_TIMESTAMP

DBPARTITIONNAME	ENCRYPT_TDES	RAND	VARCHAR_FORMAT
DECRYPT_BINARY	GENERATE_UNIQUE	REGEXP_COUNT	WEEK_ISO
DECRYPT_BIT	GETHINT	REGEXP_INSTR	XMLPARSE
DECRYPT_CHAR	IDENTITY_VAL_LOCAL	REGEXP_REPLACE	XMLVALIDATE
DECRYPT_DB	INSERT	REGEXP_SUBSTR	XSLTRANSFORM
DIFFERENCE	LOCATE_IN_STRING	REPEAT	

[1] For DataLinks with an attribute of FILE LINK CONTROL and READ PERMISSION DB.

column-name
Names a column of the index. Do not use the same name for more than one column of the index or for a *system-column-name* of the index.

If the *expression* is not a column name and is not named, a name will be generated for the index key column. The name will be SQLIXxxxxx, where *xxxxx* is a number that makes the column name unique for the index.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. Do not use the same name for more than one column of the index or for a column-name of the index.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. The name will be SQLIXxxxxx, where *xxxxx* is a number that makes the column name unique for the index.

ASC
Specifies that the index entries are to be kept in ascending order of the column values. ASC is the default.

DESC
Specifies that the index entries are to be kept in descending order of the column values.

Ordering is performed in accordance with the comparison rules described in "Assignments and comparisons" on page 98. The null value is higher than all other values.

WHERE *search-condition*
Specifies the condition to apply for a row to be included in the index. The *search-condition* cannot contain a predicate with a subquery. It must not contain any of the items listed as restrictions for *key-expression*.

WITH *integer* **DISTINCT VALUES**
Specifies the estimated number of distinct key values. This clause may be specified for any type of index.

For encoded vector indexes this is used to determine the initial size of the codes assigned to each distinct key value. The default value is 256.

For non-encoded vector indexes, this clause is ignored.

PARTITIONED
Specifies that an index partition should be created for each data partition defined for the table using the specified columns. The *table-name* must identify a partitioned table. If the index is unique, the columns of the index must be the same or a superset of the columns of the data partition key. PARTITIONED is the default if the index is not unique and the table is partitioned.

NOT PARTITIONED
Specifies that a single index should be created that spans all of the data partitions defined for the table. The *table-name* must identify a partitioned

table. NOT PARTITIONED is the default if the index is unique and the table is partitioned. An index on a table that is not partitioned is also by default not partitioned.

If an encoded vector index is specified, NOT PARTITIONED is not allowed.

PAGESIZE
Specifies the logical page used for the index in kilobytes. Indexes with larger logical page sizes are typically more efficient when scanned during query processing. Indexes with smaller logical page sizes are typically more efficient for simple index probes and individual key look ups.

The default value for PAGESIZE is determined by the length of the key and has a minimum value of 64.

If an encoded vector index is specified, PAGESIZE is not allowed.

INCLUDE
Specifies aggregate function expressions to be included in the index. These aggregates make it possible for the index to be used directly to return aggregate results for a query. INCLUDE is only allowed for an encoded vector index.

aggregate-function-name (expression)
The aggregate function name must be one of the built-in functions AVG, COUNT, COUNT_BIG, SUM, STDDEV, STDDEV_SAMP, VARIANCE, or VARIANCE_SAMP. The *expression* argument of the aggregate function must not contain any of the items listed as restrictions for *key-expression*.

RCDFMT *format-name*
An unqualified name that designates the IBM i record format name of the index. A *format-name* is a system identifier.

If a record format name is not specified:

- If no key columns are expressions and no key columns have been explicitly named using the AS clause, the index will share the format of *table-name*.
- Otherwise, the index will not share the format of *table-name* and the *format-name* is the same as the *system-object-name* of the index.

If the INCLUDE keyword is specified, RCDFMT is not allowed.

ADD ALL COLUMNS
Specifies that all non-hidden columns of *table-name* will be added to the format for the index. All the columns will be defined in the same order as they appear in the format of *table-name* and will precede any expressions defined as index keys.

ADD KEYS ONLY
Specifies that only the columns specified as index key columns will be added to the format for the index. Other columns from *table-name* will not be added.

ADD *column-name*
Specifies that the listed columns will be added to the format for the index. The index key columns will be first, followed by the added columns.

media-preference

Specifies the preferred storage media for the index.

UNIT ANY
No storage media is preferred. Storage for the index will be allocated from any available storage media.

UNIT SSD
Solid state disk storage media is preferred. Storage for the index may be allocated from solid state disk storage media, if available.

Notes

Effects of the statement: CREATE INDEX creates a description of the index. If the named table already contains data, CREATE INDEX creates the index entries for it. If the table does not yet contain data, the index entries are created when data is inserted into the table.

Collating sequence: Any index created over columns containing SBCS or mixed data is created with the collating sequence in effect at the time the statement is executed. For collating sequences other than *HEX, the key for SBCS data or mixed data is the weighted value of the key based on the collating sequence.

Index attributes: An index is created as a keyed logical file. When an index is created, the file wait time and record wait time attributes are set to the default that is specified on the WAITFILE and WAITRCD keywords of the Create Logical File (CRTLF) command.

An index created over a distributed table is created on all of the servers across which the table is distributed. For more information about distributed tables, see DB2 Multisystem.

Index ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the index is created exists, the *owner* of the index is that user profile.
- Otherwise, the *owner* of the index is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the index is the user profile or group user profile of the job executing the statement.

Index authority: If SQL names are used, indexes are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, indexes are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the index is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the index.

Examples

Example 1: Create an index named UNIQUE_NAM on the PROJECT table. The purpose of the index is to ensure that there are not two entries in the table with the same value for project name (PROJNAME). The index entries are to be in ascending order.

```
CREATE UNIQUE INDEX UNIQUE_NAM
  ON PROJECT(PROJNAME)
```

Example 2: Create an index named JOB_BY_DPT on the EMPLOYEE table. Arrange the index entries in ascending order by job title (JOB) within each department (WORKDEPT).

```
CREATE INDEX JOB_BY_DPT
  ON EMPLOYEE (WORKDEPT, JOB)
```

Example 3: Create an index named DEPT_TYPE on the DEPARTMENT table. Arrange the index entries in ascending order by type of department, which is determined by the second and third characters of the department number (DEPTNO).

```
CREATE INDEX DEPT_TYPE
 ON DEPARTMENT (SUBSTR(DEPTNO,2,2))
```

CREATE PROCEDURE

The CREATE PROCEDURE statement defines a procedure at the current server.

The following types of procedures can be defined:

- External

  The procedure program or service program is written in a programming language such as C, COBOL, or Java. The external executable is referenced by a procedure defined at the current server along with various attributes of the procedure. See "CREATE PROCEDURE (External)" on page 939.
- SQL

  The procedure is written exclusively in SQL. The procedure body is defined at the current server along with various attributes of the procedure. See "CREATE PROCEDURE (SQL)" on page 955.

Notes

Choosing data types for parameters: For portability of procedures across platforms that are not DB2 for i, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying AS LOCATOR for a parameter: Passing a locator instead of a value can result in fewer bytes being passed in or out of the procedure. This can be useful when the value of the parameter is very large. The AS LOCATOR clause specifies that a locator to the value of the parameter is passed instead of the actual value. Specify AS LOCATOR only for parameters with a LOB or XML data type or a distinct type based on a LOB or XML data type.

AS LOCATOR cannot be specified for SQL procedures.

Determining the uniqueness of procedures in a schema: At the current server, each procedure signature must be unique. The signature of a procedure is the qualified procedure name combined with the number of the parameters (the data types of the parameters are not part of a procedure's signature). This means that two different schemas can each contain a procedure with the same name that have the same number of parameters. However, a schema must not contain two procedures with the same name that have the same number of parameters.

The specific name for a procedure: When defining multiple procedures with the same name and schema (with different number of parameters), it is recommended that a specific name also be specified. The specific name can be used to uniquely identify the procedure when dropping, granting to, revoking from, or commenting on the procedure.

If *specific-name* is not specified, it is the same as the procedure name. If a function or procedure with that specific name already exists, a unique name is generated similar to the rules used to generate unique table names.

Special registers in procedures: The settings of the special registers of the caller are inherited by the procedure when called and restored upon return to the caller. Special registers may be changed within a procedure, but these changes do not affect the caller.

Restore considerations: When a procedure's associated program or service program is saved and subsequently restored and the object was updated with the procedure attributes when the procedure was created, the saved attributes will be processed and possibly changed during the restore.

If the 'Saved library' (SAVLIB) of the program or service program is different from the 'Restore to library' (RSTLIB), the procedure's schema name, specific schema name, and the external name may be changed as a result of the restore.

- If the saved procedure schema name and the library name of the saved object match, the procedure schema will be changed to the 'Restore to library' (RSTLIB). Otherwise, the procedure schema name is the saved procedure schema name.
- The specific schema name is always the same as procedure schema name.
- If the saved EXTERNAL NAME library and the library name of the saved object match, the EXTERNAL NAME library will be changed to the 'Restore to library' (RSTLIB). Otherwise, the EXTERNAL NAME library is the saved library name. If the saved EXTERNAL NAME library is *LIBL, it will not change.

If the same procedure name and number of parameters already exists in the catalog:

- If the external program name or service program name is the same as the one that already exists in the catalog, the information in the catalog for that procedure will be replaced with the saved attributes (including the specific name).
- Otherwise, the saved attributes are not restored, and a warning (SQL9015) is issued.

If the same specific name already exists in the catalog, a warning is issued and a new specific name is generated. Otherwise, the specific name of the procedure is preserved.

CREATE PROCEDURE (External)

The CREATE PROCEDURE (External) statement defines an external procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSPROCS catalog view and SYSPARMS catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

If the external program or service program exists, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the external program or service program that is referenced in the SQL statement:
  - The system authority *EXECUTE on the library that contains the external program or service program.
  - The system authority *EXECUTE on the external program or service program, and
  - The system authority *CHANGE on the program or service program. The system needs this authority to update the program or service program object to contain the information necessary to save/restore the procedure to another system. If user does not have this authority, the procedure is still created, but the program or service program object is not updated.
- Administrative Authority

If a distinct type or array type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

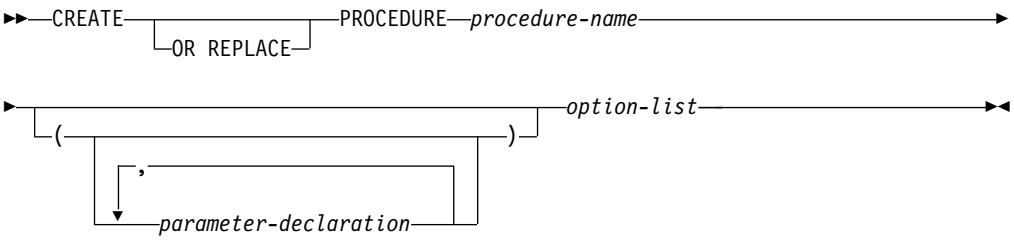
- For each distinct type or array type identified in the statement:
  - The USAGE privilege on the type, and
  - The system authority *EXECUTE on the library containing the distinct type or array type
- Administrative authority

To replace an existing procedure, the privileges held by the authorization ID of the statement must include at least one of the following:

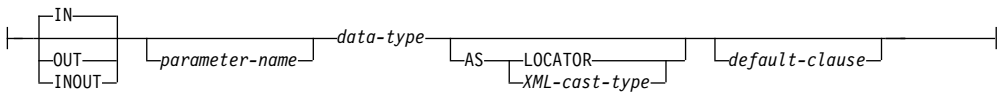
- The following system authorities:
  - The system authority of *OBJMGT on the program object associated with the procedure
  - All authorities needed to DROP the procedure
  - The system authority *READ to the SYSPROCS catalog view and SYSPARMS catalog table
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure and Corresponding System Authorities When Checking Privileges to a Distinct Type.

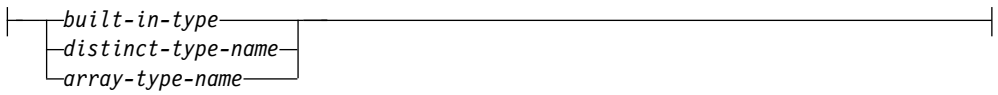
Syntax



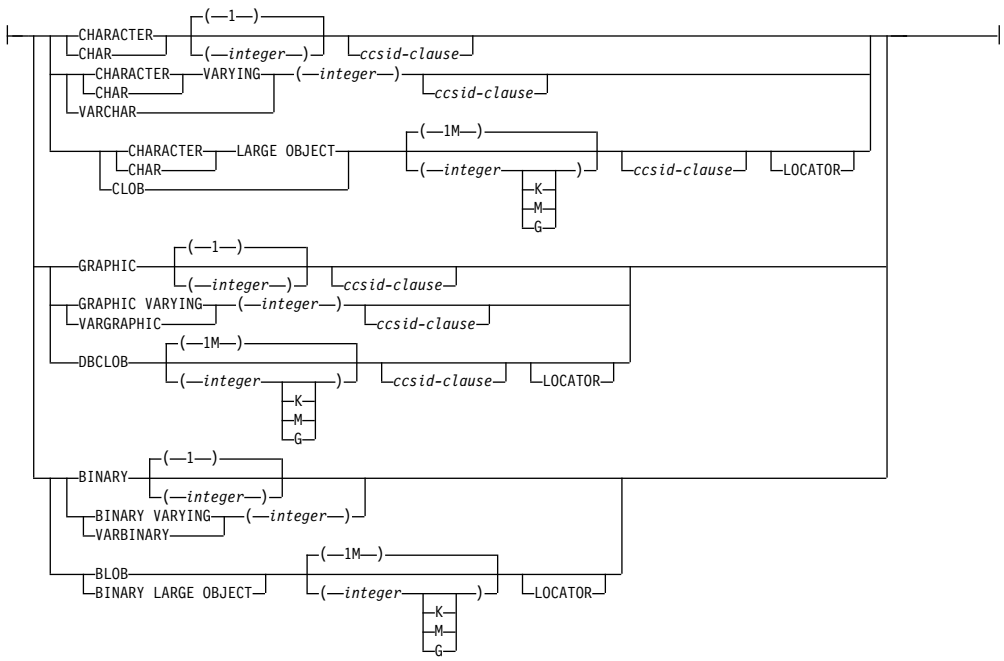
parameter-declaration:



data-type:

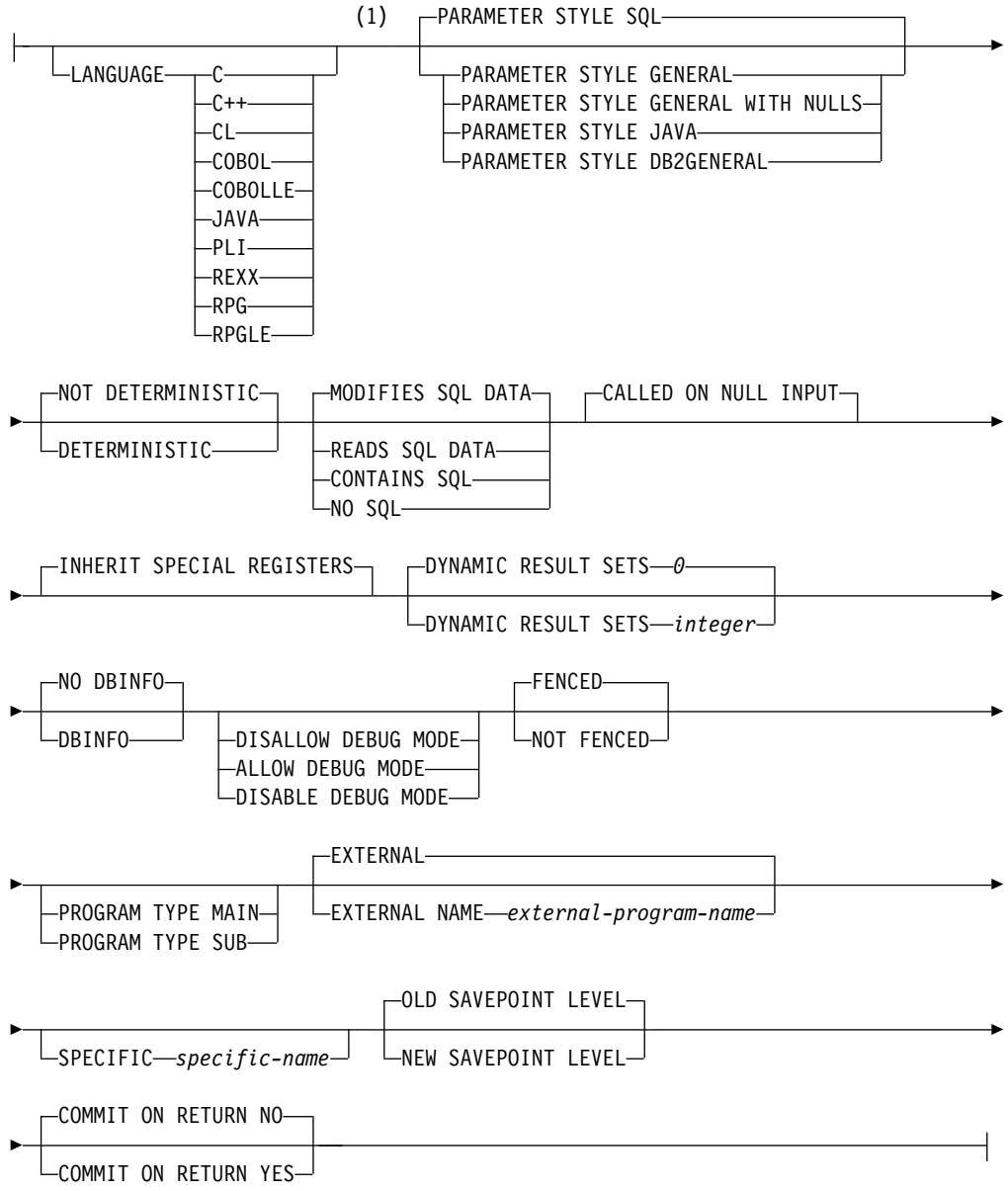


XML-cast-type:



default-clause:

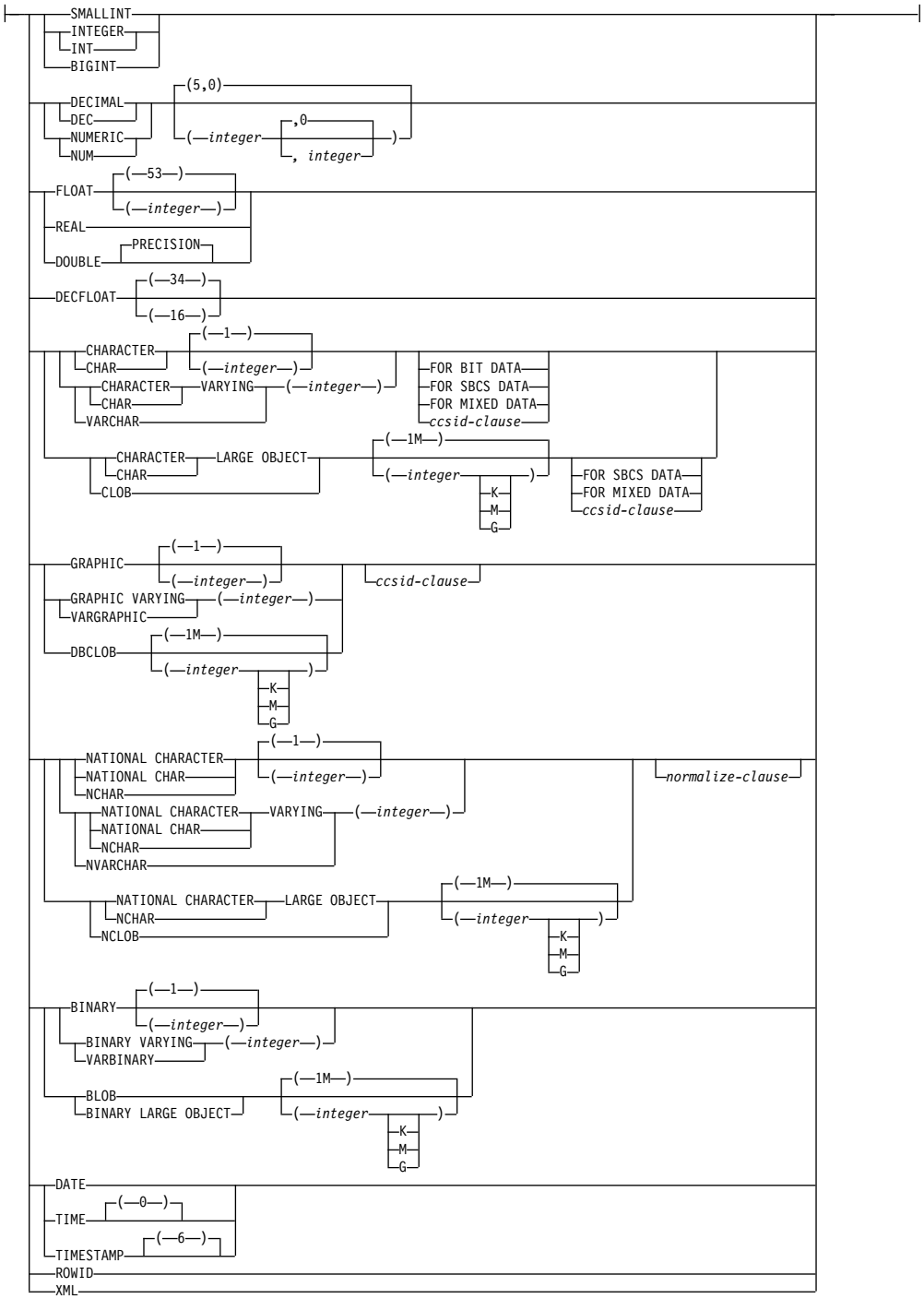
option-list:



Notes:

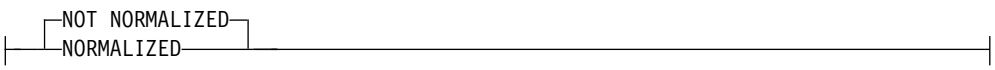
1 The optional clauses can be specified in a different order.

built-in-type:



ccsid-clause:

normalize-clause:



Description

OR REPLACE
Specifies to replace the definition for the procedure if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog with the exception that privileges that were granted on the procedure are not affected. This option is ignored if a definition for the procedure does not exist at the current server. To replace an existing procedure, the *specific-name* and *procedure-name* of the new definition must be the same as the *specific-name* and *procedure-name* of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new procedure is created.

procedure-name
Names the procedure. The combination of name, schema name, the number of parameters must not identify a procedure that exists at the current server.

For SQL naming, the procedure will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the procedure will be created in the schema specified by the qualifier. If no qualifier is specified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the procedure will be created in the current library (*CURLIB).
- Otherwise, the procedure will be created in the current schema.

(*parameter-declaration***,...)**
Specifies the number of parameters of the procedure and the data type of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. Although not required, you can give each parameter a name.

The maximum number of parameters allowed in CREATE PROCEDURE depends on the language and the parameter style:

- If PARAMETER STYLE GENERAL is specified, in C and C++, the maximum is 1024. Otherwise, the maximum is 255.
- If PARAMETER STYLE GENERAL WITH NULLS is specified, in C and C++, the maximum is 1023. Otherwise, the maximum is 254.
- If PARAMETER STYLE SQL is specified, in C and C++, the maximum is 508. Otherwise, the maximum is 90.
- If PARAMETER STYLE JAVA or PARAMETER STYLE DB2GENERAL is specified, the maximum is 90.

The maximum number of parameters is also limited by the maximum number of parameters allowed by the licensed program used to compile the external program or service program.

IN Identifies the parameter as an input parameter to the procedure. Any changes made to the parameter within the procedure are not available to the calling SQL application when control is returned.[93] The default is IN.

93. When the language type is REXX, all parameters must be input parameters.

OUT
Identifies the parameter as an output parameter that is returned by the procedure.

A DataLink or a distinct type based on a DataLink cannot be specified as an output parameter.

INOUT
Identifies the parameter as both an input and output parameter for the procedure. If an INOUT parameter is defined with a default and the default is used when calling the procedure, no value for the parameter is returned.

A DataLink or a distinct type based on a DataLink cannot be specified as an input and output parameter.

parameter-name
Names the parameter. The name cannot be the same as any other *parameter-name* for the procedure.

data-type
Specifies the data type of the parameter.

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see "CREATE TABLE" on page 982.

distinct-type-name
Specifies a distinct type. Any length, precision, scale, or encoding scheme attributes for the parameter are those of the source type of the distinct type as specified using "CREATE TYPE (Distinct)" on page 1055.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

array-type-name
Specifies an array type. Array types are only supported for LANGUAGE JAVA. To use an array type as a parameter for a Java stored procedure, the parameter style must be JAVA.

If the name of the array type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

If a CCSID is specified, the parameter will be converted to that CCSID before passing it to the procedure. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the procedure is invoked.

Some data types are not supported in all languages. For details on the mapping between the SQL data types and host language data types, see Embedded SQL Programming topic collection. Built-in data type specifications can be specified if they correspond to the language that is used to write the procedure.

Any parameter that has an XML type must specify either the *XML-cast-type* clause or the AS LOCATOR clause.

AS LOCATOR
Specifies that the parameter is a locator to the value rather than the actual value. You can specify AS LOCATOR only if the parameter has a LOB or

XML data type or a distinct type based on a LOB or XML data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

AS *XML-cast-type*
Specifies the data type passed to the procedure for a parameter that is XML type or a distinct type based on XML type. If LOCATOR is specified, the parameter is a locator to the value rather than the actual value.

If a CCSID value is specified, only Unicode CCSID values can be specified for graphic data types. If a CCSID value is not specified, the CCSID is established at the time the procedure is created according to the SQL_XML_DATA_CCSID QAQQINI option setting. The default CCSID is 1208. See "XML Values" on page 88 for a description of this option.

default
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The expression is any expression defined in "Expressions" on page 165, that does not include an aggregate function or column name. If a default value is not specified, the parameter has no default and cannot be omitted on invocation of the procedure. The maximum length of the expression string is 64K.

The default expression must not modify SQL data. The expression must be assignment compatible to the parameter data type. All objects referenced in a default expression must exist when the procedure is created. When the procedure is called, the default will be evaluated using the authority of the invoker.

Any comma in the default expression that is intended as a separator of numeric constants in a list must be followed by a space.

A default cannot be specified:

- for an OUT parameter.
- for a parameter of type array.

LANGUAGE
Specifies the language that the external program or service program is written in. The language clause is required if the external program is a REXX procedure.

If LANGUAGE is not specified, the LANGUAGE is determined from the attribute information associated with the external program or service program at the time the procedure is created. If the attribute information associated with the program or service program does not identify a recognizable language or the program or service program cannot be found, then the language is assumed to be C.

C The external program is written in C.

C++
The external program is written in C++.

CL The external program is written in CL.

COBOL
The external program is written in COBOL.

COBOLLE
The external program is written in ILE COBOL.

JAVA
The external program is written in JAVA.

PLI
The external program is written in PL/I.

REXX
The external program is a REXX procedure.

RPG
The external program is written in RPG.

RPGLE
The external program is written in ILE RPG.

PARAMETER STYLE
Specifies the conventions used to pass parameters to and returning the values from procedures:

SQL
Specifies that in addition to the parameters on the CALL statement, several additional parameters are passed to the procedure. The parameters are defined to be in the following order:

- The first *n* parameters are the parameters that are specified on the CREATE PROCEDURE statement.
- *n* parameters for indicator variables for the parameters.
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the procedure. The SQLSTATE returned is assigned by the external program.

  The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the procedure.
- A VARCHAR(517) input parameter for the fully qualified procedure name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(1000) output parameter for the message text.

The following additional parameter may be passed as the last parameter:

- A parameter for the dbinfo structure, if DBINFO was specified on the CREATE PROCEDURE statement.

These parameters are passed according to the specified LANGUAGE. For example, if the language is C or C++, the VARCHAR parameters are passed as NUL-terminated strings. For more information about the parameters passed, see the include sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

PARAMETER STYLE SQL cannot be used with LANGUAGE JAVA.

DB2GENERAL
Specifies that the procedure will use a parameter passing convention that is defined for use with Java methods.

PARAMETER STYLE DB2GENERAL can only be specified with LANGUAGE JAVA. For details on passing parameters in JAVA, see the IBM Developer Kit for Java.

GENERAL
Specifies that the procedure will use a parameter passing mechanism

where the procedure receives the parameters specified on the CALL. Additional arguments are not passed for indicator variables.

PARAMETER STYLE GENERAL cannot be used with LANGUAGE JAVA.

GENERAL WITH NULLS
Specifies that in addition to the parameters on the CALL statement as specified in GENERAL, another argument is passed to the procedure. This additional argument contains an indicator array with an element for each of the parameters of the CALL statement. In C, this would be an array of short INTs. For more information about how the indicators are handled, see the SQL Programming topic collection.

PARAMETER STYLE GENERAL WITH NULLS cannot be used with LANGUAGE JAVA.

JAVA
Specifies that the procedure will use a parameter passing convention that conforms to the Java language and ISO/IEC FCD 9075-13:2003, *Information technology - Database languages - SQL - Part 13: Java Routines and Types (SQL/JRT)* specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values.

PARAMETER STYLE JAVA can only be specified with LANGUAGE JAVA. For increased portability, you should write Java procedures that use the PARAMETER STYLE JAVA conventions. For details on passing parameters in JAVA, see the IBM Developer Kit for Java topic collection.

Note that the language of the external procedure determines how the parameters are passed. For example, in C, any VARCHAR or CHAR parameters are passed as NUL-terminated strings. For more information, see the SQL Programming topic collection. For Java routines, see the IBM Developer Kit for Java topic collection.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
The procedure may not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

DETERMINISTIC
The procedure always returns the same results each time the procedure is called with the same IN and INOUT arguments, provided the referenced data in the database has not changed.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL
Specifies the classification of SQL statements that this procedure, or any routine called by this procedure, can execute. The database manager verifies that the SQL statements issued by the procedure and all routines called by the procedure are consistent with this specification. The default is MODIFIES SQL DATA. This option is ignored for parameter default expressions. For the classification of each statement, see Appendix B, "Characteristics of SQL statements," on page 1501.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA
Specifies that the procedure can execute statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL.

CONTAINS SQL
Specifies that the procedure can only execute statements with a data access classification of CONTAINS SQL or NO SQL.

NO SQL
Specifies that the procedure can execute only SQL statements with a data access classification of NO SQL.

CALLED ON NULL INPUT
Specifies that the procedure is to be called if any or all argument values are null. This specification means that the procedure must be coded to test for null argument values.

INHERIT SPECIAL REGISTERS
Specifies that existing values of special registers are inherited upon entry to the procedure.

FENCED or NOT FENCED
This parameter is allowed for compatibility with other products and is not used by DB2 for i.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of result sets that can be returned from the procedure. The minimum value for *integer* is zero and the maximum value is 32768.

If no DYNAMIC RESULT SETS clause is specified, result sets are returned for all cursors that remain open when the procedure ends.

Result sets are returned in the order in which the corresponding cursors are opened, unless a SET RESULT SETS statement is executed in the procedure. If the number of cursors that are still open for result sets when the procedure ends exceeds the maximum number specified on the DYNAMIC RESULT SETS clause, a warning is returned on the CALL statement and the number of result sets specified on the DYNAMIC RESULT SETS clause is returned.

If the SET RESULT SETS statement is issued, the number of results returned is the minimum of the number of result sets specified on this keyword and the SET RESULT SETS statement. If the SET RESULT SETS statement specifies a number larger than the maximum number of result sets, a warning is returned. Note that any result sets from cursors that have a RETURN TO CLIENT attribute are included in the number of result sets of the outermost procedure.

The result sets are scrollable if a cursor is used to return a result set and the cursor is scrollable. If a cursor is used to return a result set, the result set starts with the current position. Thus, if 5 FETCH NEXT operations have been performed before returning from the procedure, the result set starts with the 6th row of the result set.

| Cursor result sets are only returned if the external program does not have an
| attribute of ACTGRP(*NEW).

For more information about result sets, see "SET RESULT SETS" on page 1390.

DISALLOW DEBUG MODE, ALLOW DEBUG MODE, or DISABLE DEBUG MODE
Indicates whether the procedure is created so it can be debugged by the

Unified Debugger. If DEBUG MODE is not specified, the procedure will be created with the debug mode specified by the CURRENT DEBUG MODE special register.

DEBUG MODE can only be specified with LANGUAGE JAVA.

DISALLOW DEBUG MODE
The procedure cannot be debugged by the Unified Debugger. When the DEBUG MODE attribute of the procedure is DISALLOW, the procedure can be subsequently altered to change the debug mode attribute.

ALLOW DEBUG MODE
The procedure can be debugged by the Unified Debugger. When the DEBUG MODE attribute of the procedure is ALLOW, the procedure can be subsequently altered to change the debug mode attribute.

DISABLE DEBUG MODE
The procedure cannot be debugged by the Unified Debugger. When the DEBUG MODE attribute of the procedure is DISABLE, the procedure cannot be subsequently altered to change the debug mode attribute.

PROGRAM TYPE
This parameter is allowed for compatibility with other products. It indicates whether the routine's external program is a program (*PGM) or a procedure in a service program (*SRVPGM).

SUB
Specifies that the procedure executes as a procedure in a service program. The external program must be a *SRVPGM object.

MAIN
Specifies that the routine executes as the main entry point in a program. The external program must be a *PGM object.

DBINFO
Specifies whether additional status information is passed to the procedure when it is called. The default is NO DBINFO.

NO DBINFO
Additional information is not passed.

DBINFO
An additional argument is passed when the procedure is called.

The argument is a structure that contains information such as the name of the current server, the application run-time authorization ID, and identification of the version and release of the database manager that called the procedure. See Table 76 for further details. Detailed information about the DBINFO structure can be found in include sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

DBINFO is only allowed with PARAMETER STYLE SQL.

Table 76. DBINFO fields

Field	Data Type	Description
Relational database	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The run-time authorization ID.

Table 76. DBINFO fields (continued)

Field	Data Type	Description
CCSID information	INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER CHAR(8)	The CCSID information of the job. Three sets of three CCSIDs are returned. The following information identifies the three CCSIDs in each set: • SBCS CCSID • DBCS CCSID • Mixed CCSID Following the three sets of CCSIDs is an integer that indicates which set of three sets of CCSIDs is applicable and 8 bytes of reserved space. Each set of CCSIDs is for a different encoding scheme (EBCDIC, ASCII, and Unicode). If a CCSID is not explicitly specified for a parameter on the CREATE PROCEDURE statement, the input string is assumed to be encoded in the CCSID of the job at the time the procedure is executed. If the CCSID of the input string is not the same as the CCSID of the parameter, the input string passed to the external procedure will be converted before calling the external program.
Target column	VARCHAR(128) VARCHAR(128) VARCHAR(128)	Not applicable for a call to a procedure.
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server's platform type.

EXTERNAL

Specifies that the CREATE PROCEDURE statement is being used to define a new procedure based on code written in an external programming language.

If NAME clause is not specified, "NAME *procedure-name*" is assumed. In this case, *procedure-name* must not be longer than 8 characters. The NAME clause is required for a LANGUAGE JAVA procedure since the default name is not valid for a Java procedure.

NAME *external-program-name*

Specifies the program or service program that will be executed when the procedure is called by the CALL statement. The program name must identify a program or service program that exists at the application server at the time the procedure is called. If the naming option is *SYS and the name is not qualified:

- The current path will be used to search for the program at the time the procedure is called.
- *LIBL will be used to search for the program or service program at the time COMMENT, GRANT, LABEL, or REVOKE operations are performed on the procedure.

The validity of the name is checked at the application server. If the format of the name is not correct, an error is returned.

The external program or service program need not exist at the time the procedure is created, but it must exist at the time the procedure is called.

The SET TRANSACTION statement is not allowed in a procedure that is running on a remote application server. COMMIT and ROLLBACK statements are not allowed in an ATOMIC SQL procedure or in a procedure that is running on a connection to a remote application server.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a procedure.

OLD SAVEPOINT LEVEL or NEW SAVEPOINT LEVEL
Specifies whether a new savepoint level is to be created on entry to the procedure.

OLD SAVEPOINT LEVEL
A new savepoint level is not created. Any SAVEPOINT statements issued within the procedure with OLD SAVEPOINT LEVEL implicitly or explicitly specified on the SAVEPOINT statement are created at the same savepoint level as the caller of the procedure. This is the default.

NEW SAVEPOINT LEVEL
A new savepoint level is created on entry to the procedure. Any savepoints set within the procedure are created at a savepoint level that is nested deeper than the level at which this procedure was invoked. Therefore, the name of any new savepoint set within the procedure will not conflict with any existing savepoints set in higher savepoint levels (such as the savepoint level of the calling program or service program) with the same name.

COMMIT ON RETURN
Specifies whether the database manager commits the transaction immediately on return from the procedure.

NO The database manager does not issue a commit when the procedure returns. NO is the default.

YES
The database manager issues a commit if the procedure returns successfully. If the procedure returns with an error, a commit is not issued.

The commit operation includes the work that is performed by the calling application process and the procedure.[94]

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

SET OPTION-statement
Specifies the options that will be used for parameter defaults. The default values for the options depend on the options in effect at create time. For more information, see "SET OPTION" on page 1368.

The following options are used when processing default value expressions: ALWCPYDTA, CONACC, DATFMT, DATSEP, DECFLTRND, DECMPT,

94. If the external program or service program was created with ACTGRP(*NEW) and the job commitment definition is not used, the work that is performed in the procedure will be committed or rolled back as a result of the activation group ending.

DECRESULT, DFTRDBCOL, LANGID, SQLCURRULE, SQLPATH, SRTSEQ, TGTRLS, TIMFMT, and TIMSEP. The options CNULRQD, CNULIGN, COMPILEOPT, EXTIND, NAMING, and SQLCA are not allowed in the CREATE PROCEDURE statement. Other options are accepted but will be ignored.

Notes

General considerations for defining procedures: See "CREATE PROCEDURE" on page 937 for general information about defining procedures.

Language considerations: For information needed to create the programs for a procedure, see Embedded SQL Programming.

REPLACE rules: When an external procedure is recreated by REPLACE:

- Any existing comment or label is discarded.
- If a different external program is specified:
  - Authorized users are not copied to the new program.
  - Journal auditing is not changed.
- Otherwise:
  - Authorized users are maintained. The object owner will not change.
  - Current journal auditing is not changed.

Error handling considerations: Values of arguments passed to a procedure which correspond to OUT parameters are undefined and those which correspond to INOUT parameters are unchanged when an error is returned by the procedure.

Creating the procedure: When an external procedure associated with an ILE external program or service program is created, an attempt is made to save the procedure's attributes in the associated program or service program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the attributes are used to update the catalogs.

The attributes can be saved for external procedures subject to the following restrictions:

- The external program library must not be QSYS.
- The external program must exist when the CREATE PROCEDURE statement is issued.

  If system naming is specified and the external program name is not qualified, the external program must be found in the library list.
- The external program must be an ILE *PGM or *SRVPGM object.
- The external program must not already contain attributes for 32 routines.

If the object cannot be updated, the procedure will still be created.

Calling the procedure: If a DECLARE PROCEDURE statement defines a procedure with the same name as a created procedure, and a static CALL statement where the procedure name is not identified by a variable is executed from the same source program, the attributes from the DECLARE PROCEDURE statement will be used rather than the attributes from the CREATE PROCEDURE statement.

The CREATE PROCEDURE statement applies to static and dynamic CALL statements as well as to a CALL statement where the procedure name is identified by a variable.

When an external procedure is invoked, it runs in whatever activation group was specified when the external program or service program was created. However, ACTGRP(*CALLER) should normally be used so that the procedure runs in the same activation group as the calling program.

Setting of the default value: Parameters of a procedure that are defined with a default value are set to their default value when the procedure is invoked, but only if a value is not supplied for the corresponding argument, or the argument is specified as DEFAULT.

Notes for Java procedures: To be able to run Java procedures, you must have the IBM Developer Kit for Java installed on your system. Otherwise, an SQLCODE of -443 will be returned and a CPDB521 message will be placed in the job log.

If an error occurs while running a Java procedure, an SQLCODE of -443 will be returned. Depending on the error, other messages may exist in the job log of the job where the procedure was run.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL can be used as synonyms for CALLED ON NULL INPUT.
- The keywords SIMPLE CALL can be used as a synonym for GENERAL.
- The value DB2GENRL may be used as a synonym for DB2GENERAL.
- DYNAMIC RESULT SET, RESULT SETS, and RESULT SET may be used as synonyms for DYNAMIC RESULT SETS.
- The keywords PARAMETER STYLE in the PARAMETER STYLE clause are optional.
- The keywords PARAMETER STYLE DB2SQL can be used as a synonym for PARAMETER STYLE SQL.

Example

Example 1: Create the procedure definition for a procedure, written in Java, that is passed a part number and returns the cost of the part and the quantity that are currently available.

```
CREATE PROCEDURE PARTS_ON_HAND (IN  PARTNUM  INTEGER,
                                OUT COST     DECIMAL(7,2),
                                OUT QUANTITY INTEGER)
       LANGUAGE JAVA
       PARAMETER STYLE JAVA
       EXTERNAL NAME 'parts.onhand'
```

Example 2: Create the procedure definition for a procedure, written in C, that is passed an assembly number and returns the number of parts that make up the assembly, total part cost and a result set that lists the part numbers, quantity and unit cost of each part.

```
CREATE PROCEDURE ASSEMBLY_PARTS (IN  ASSEMBLY_NUM  INTEGER,
                                 OUT NUM_PARTS     INTEGER,
                                 OUT COST DOUBLE)
       LANGUAGE C
       PARAMETER STYLE GENERAL
       DYNAMIC RESULT SETS 1
       FENCED
       EXTERNAL NAME ASSEMBLY
```

CREATE PROCEDURE (SQL)

The CREATE PROCEDURE (SQL) statement creates an SQL procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSPROCS catalog view and SYSPARMS catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - *USE on the Create Program (CRTPGM) command, and
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the procedure is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

If a distinct type or array type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type or array type identified in the statement:
  - The USAGE privilege on the type, and
  - The system authority *EXECUTE on the library containing the distinct type or array type
- Administrative authority

To replace an existing procedure, the privileges held by the authorization ID of the statement must include at least one of the following:

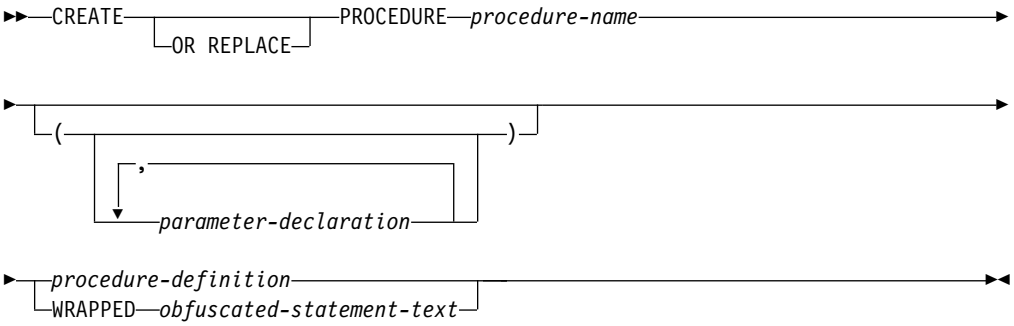
- The following system authorities:
  - The system authority of *OBJMGT on the program or service program object associated with the procedure

- All authorities needed to DROP the procedure
- The system authority *READ to the SYSPROCS catalog view and SYSPARMS catalog table

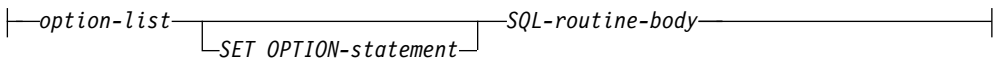
- Administrative authority

For information on the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Function or Procedure and Corresponding System Authorities When Checking Privileges to a Distinct Type.

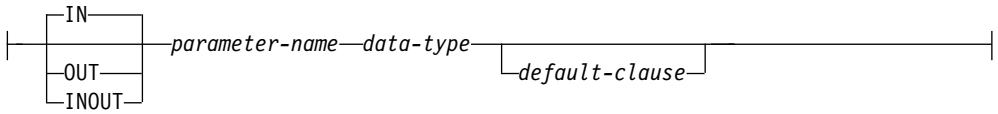
Syntax



procedure-definition:



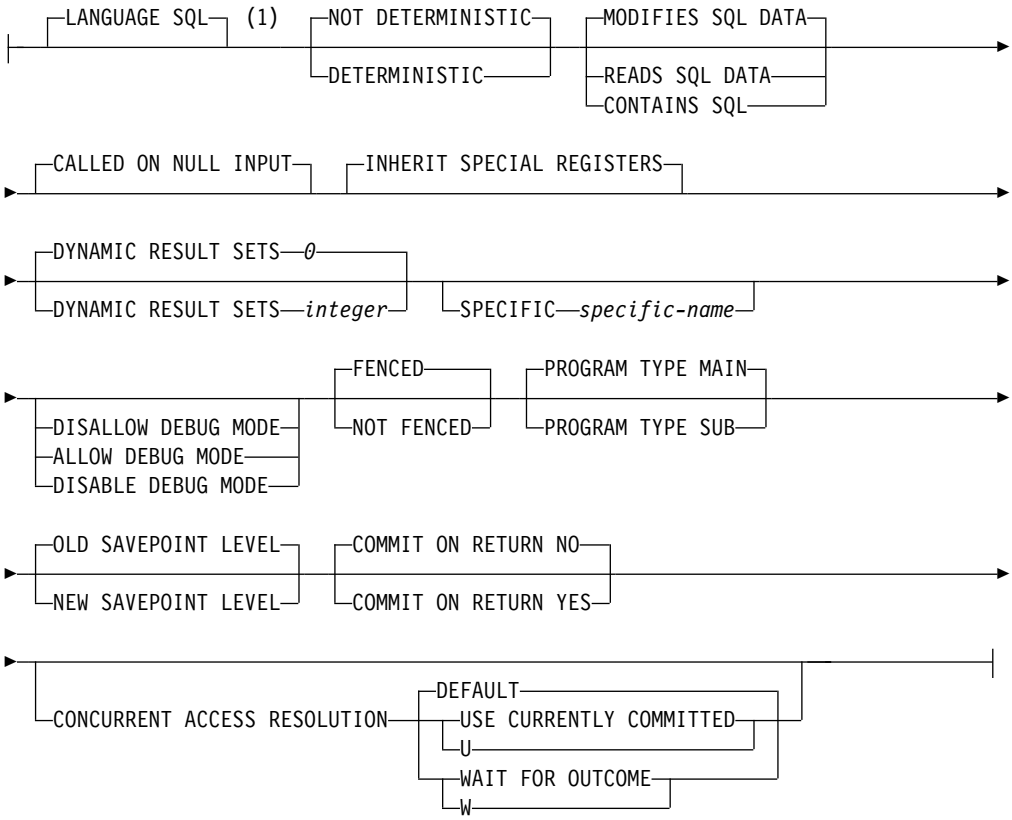
parameter-declaration:



data-type:

default-clause:

option-list:



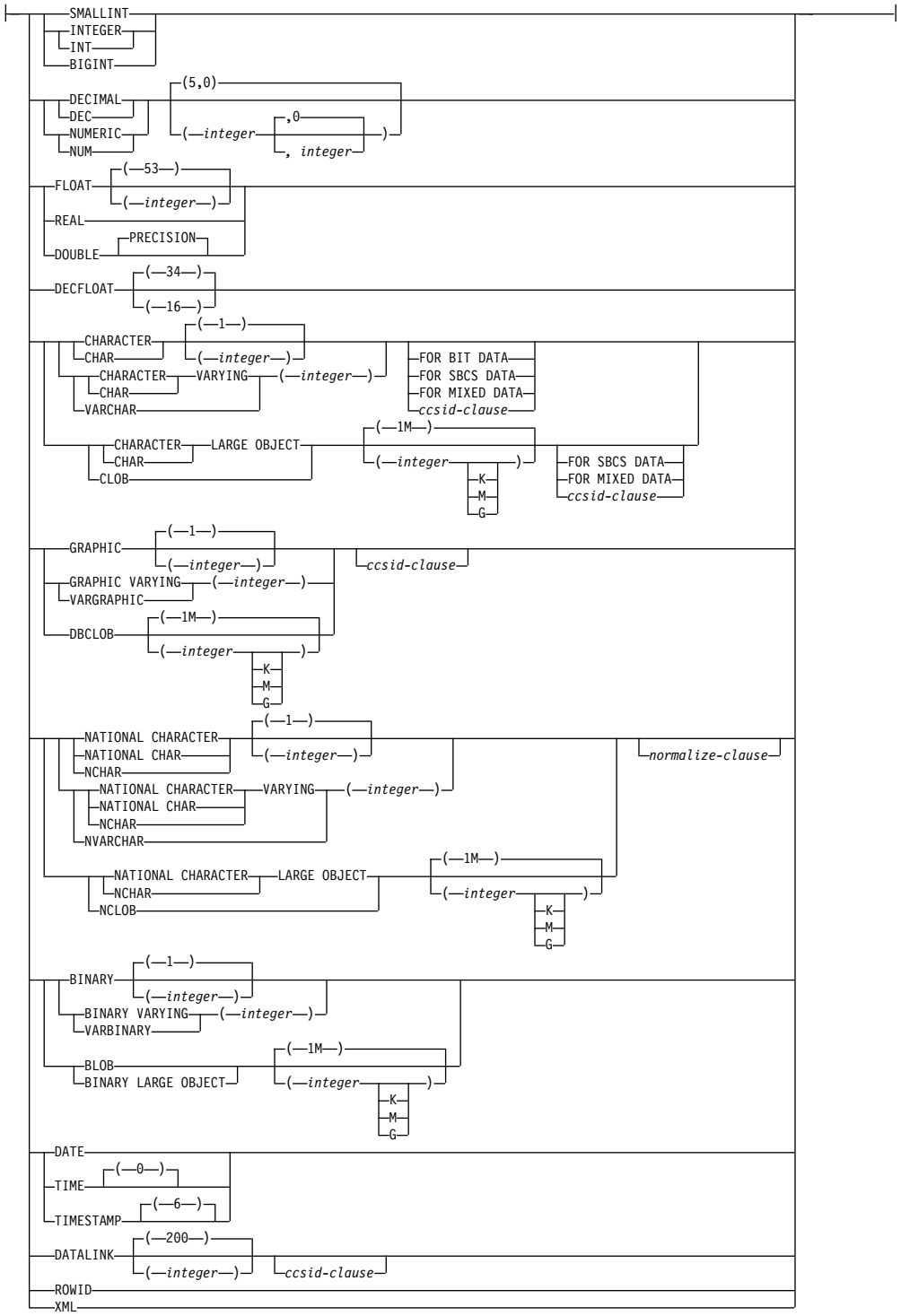
Notes:

1 The optional clauses can be specified in a different order.

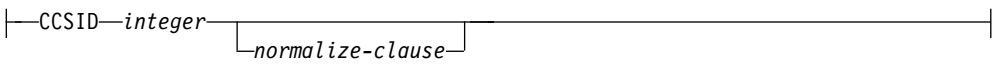
SQL-routine-body:


SQL-control-statement
ALLOCATE CURSOR-statement
ALLOCATE DESCRIPTOR-statement
ALTER FUNCTION-statement
ALTER PROCEDURE-statement
ALTER SEQUENCE-statement
ALTER TABLE-statement
ASSOCIATE LOCATORS-statement
COMMENT-statement
COMMIT-statement
CONNECT-statement
CREATE ALIAS-statement
CREATE FUNCTION (External Scalar)-statement
CREATE FUNCTION (External Table)-statement
CREATE FUNCTION (Sourced)-statement
CREATE INDEX-statement
CREATE PROCEDURE (External)-statement
CREATE SCHEMA-statement
CREATE SEQUENCE-statement
CREATE TABLE-statement
CREATE TYPE-statement
CREATE VIEW-statement
DEALLOCATE DESCRIPTOR-statement
DECLARE GLOBAL TEMPORARY TABLE-statement
DELETE-statement
DESCRIBE-statement
DESCRIBE CURSOR-statement
DESCRIBE INPUT-statement
DESCRIBE PROCEDURE-statement
DESCRIBE TABLE-statement
DISCONNECT-statement
DROP-statement
EXECUTE IMMEDIATE-statement
GET DESCRIPTOR-statement
GRANT-statement
INSERT-statement
LABEL-statement
LOCK TABLE-statement
MERGE-statement
REFRESH TABLE-statement
RELEASE-statement
RELEASE SAVEPOINT-statement
RENAME-statement
REVOKE-statement
ROLLBACK-statement
SAVEPOINT-statement
SELECT INTO-statement
SET CONNECTION-statement
SET CURRENT DEBUG MODE-statement
SET CURRENT DECFLOAT ROUNDING MODE-statement
SET CURRENT DEGREE-statement
SET CURRENT IMPLICIT XMLPARSE OPTION-statement
SET DESCRIPTOR-statement
SET ENCRYPTION PASSWORD-statement
SET PATH-statement
SET RESULT SETS-statement
SET SCHEMA-statement
SET TRANSACTION-statement
UPDATE-statement
VALUES INTO-statement

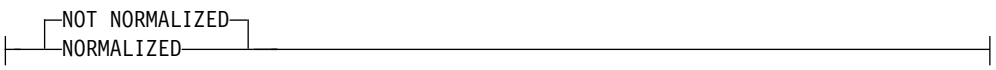
built-in-type:



ccsid-clause:



normalize-clause:



Description

OR REPLACE
Specifies to replace the definition for the procedure if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog with the exception that privileges that were granted on the procedure are not affected. This option is ignored if a definition for the procedure does not exist at the current server. To replace an existing procedure, the *specific-name* and *procedure-name* of the new definition must be the same as the *specific-name* and *procedure-name* of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new procedure is created.

procedure-name
Names the procedure. The combination of name, schema name, the number of parameters must not identify a procedure that exists at the current server.

For SQL naming, the procedure will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the procedure will be created in the schema specified by the qualifier. If no qualifier is specified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the procedure will be created in the current library (*CURLIB).
- Otherwise, the procedure will be created in the current schema.

The *schema-name* cannot be QSYS2, QSYS, QTEMP, or SYSIBM.

(*parameter-declaration***,...)**
Specifies the number of parameters of the procedure and the data type of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output.

The maximum number of parameters allowed in an SQL procedure is 1024.

IN Identifies the parameter as an input parameter to the procedure. Any changes made to the parameter within the procedure are not available to the calling SQL application when control is returned. The default is IN.

OUT
Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT
Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned. If an INOUT parameter is defined with a default and the default is used when calling the procedure, no value for the parameter is returned.

parameter-name
Names the parameter. The name cannot be the same as any other *parameter-name* for the procedure.

data-type
Specifies the data type of the parameter. The data type can be a built-in data type or a distinct data type.

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see "CREATE TABLE" on page 982.

distinct-type-name
Specifies a distinct type. The length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). For more information on creating a distinct type, see "CREATE TYPE (Distinct)" on page 1055. .

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

array-type-name
Specifies an array type.

If the name of the array type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the procedure. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the procedure is called.

default
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The expression is any expression defined in "Expressions" on page 165, that does not include an aggregate function or column name. If a default value is not specified, the parameter has no default and cannot be omitted on invocation of the procedure. The maximum length of the expression string is 64K.

The default expression must not modify SQL data. The expression must be assignment compatible to the parameter data type. All objects referenced in a default expression must exist when the procedure is created.

Any comma in the default expression that is intended as a separator of numeric constants in a list must be followed by a space.

A default cannot be specified:

- for an OUT parameter.
- for a parameter of type array.

LANGUAGE SQL
Specifies that this procedure is written exclusively in SQL.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a procedure.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
The procedure may not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

DETERMINISTIC
The procedure always returns the same results each time the procedure is called with the same IN and INOUT arguments, provided the referenced data in the database has not changed.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL
Specifies the classification of SQL statements that this procedure, or any routine called by this procedure, can execute. The database manager verifies that the SQL statements issued by the procedure and all routines called by the procedure are consistent with this specification. The default is MODIFIES SQL DATA. This option is used for parameter default expressions. For the classification of each statement, see Appendix B, "Characteristics of SQL statements," on page 1501.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA
Specifies that the procedure can execute statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL.

CONTAINS SQL
Specifies that the procedure can only execute statements with a data access classification of CONTAINS SQL or NO SQL.

CALLED ON NULL INPUT
Specifies that the procedure is to be called if any or all argument values are null. This specification means that the procedure must be coded to test for null argument values.

INHERIT SPECIAL REGISTERS
Specifies that existing values of special registers are inherited upon entry to the procedure.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of result sets that can be returned from the procedure. The minimum value for *integer* is zero and the maximum value is 32768. The default is DYNAMIC RESULT SETS 0.

Result sets are returned in the order in which the corresponding cursors are opened, unless a SET RESULT SETS statement is executed in the procedure. If the number of cursors still open for result sets when the procedure ends exceeds the maximum number specified on the DYNAMIC RESULT SETS clause, a warning is returned on the CALL statement and the number of result sets specified on the DYNAMIC RESULT SETS clause is returned.

If the SET RESULT SETS statement is issued, the number of results returned is the minimum of the number of result sets specified on this keyword and the SET RESULT SETS statement. If the SET RESULT SETS statement specifies a number larger than the maximum number of result sets, a warning is returned. Note that any result sets from cursors that have a RETURN TO CLIENT attribute are included in the number of result sets of the outermost procedure.

The result sets are scrollable if the cursor is used to return a result set and the cursor is scrollable. If a cursor is used to return a result set, the result set starts

with the current position. Thus, if 5 FETCH NEXT operations have been performed prior to returning from the procedure, the result set will start with the 6th row of the result set.

For more information about result sets, see "SET RESULT SETS" on page 1390.

DISALLOW DEBUG MODE, ALLOW DEBUG MODE, or DISABLE DEBUG MODE
Indicates whether the procedure is created so it can be debugged by the Unified Debugger. If DEBUG MODE is specified, a DBGVIEW option in the SET OPTION statement must not be specified.

DISALLOW DEBUG MODE
The procedure cannot be debugged by the Unified Debugger. When the DEBUG MODE attribute of the procedure is DISALLOW, the procedure can be subsequently altered to change the debug mode attribute.

ALLOW DEBUG MODE
The procedure can be debugged by the Unified Debugger. When the DEBUG MODE attribute of the procedure is ALLOW, the procedure can be subsequently altered to change the debug mode attribute.

DISABLE DEBUG MODE
The procedure cannot be debugged by the Unified Debugger. When the DEBUG MODE attribute of the procedure is DISABLE, the procedure cannot be subsequently altered to change the debug mode attribute.

If DEBUG MODE is not specified, but a DBGVIEW option in the SET OPTION statement is specified, the procedure cannot be debugged by the Unified Debugger, but may be debugged by the system debug facilities. If neither DEBUG MODE nor a DBGVIEW option is specified, the debug mode used is from the CURRENT DEBUG MODE special register.

FENCED or NOT FENCED
This parameter is allowed for compatibility with other products and is not used by DB2 for i.

PROGRAM TYPE MAIN or PROGRAM TYPE SUB
Specifies whether the procedure is created as a program (*PGM) or a procedure in a service program (*SRVPGM).

PROGRAM TYPE MAIN
Specifies that the procedure is created as a program (*PGM).

PROGRAM TYPE SUB
Specifies that the procedure is created as a procedure in a service program (*SRVPGM).

PROGRAM TYPE SUB procedures usually perform slightly better than PROGRAM TYPE MAIN procedures. If the number of parameters exceeds 400, PROGRAM TYPE MAIN will be used instead of PROGRAM TYPE SUB.

OLD SAVEPOINT LEVEL or NEW SAVEPOINT LEVEL
Specifies whether a new savepoint level is to be created on entry to the procedure.

OLD SAVEPOINT LEVEL
A new savepoint level is not created. Any SAVEPOINT statements issued within the procedure with OLD SAVEPOINT LEVEL implicitly or explicitly specified on the SAVEPOINT statement are created at the same savepoint level as the caller of the procedure. This is the default.

NEW SAVEPOINT LEVEL
A new savepoint level is created on entry to the procedure. Any savepoints set within the procedure are created at a savepoint level that is nested deeper than the level at which this procedure was invoked. Therefore, the name of any new savepoint set within the procedure will not conflict with any existing savepoints set in higher savepoint levels (such as the savepoint level of the calling program) with the same name.

COMMIT ON RETURN
Specifies whether the database manager commits the transaction immediately on return from the procedure.

NO The database manager does not issue a commit when the procedure returns. NO is the default.

YES
The database manager issues a commit if the procedure returns successfully. If the procedure returns with an error, a commit is not issued.

The commit operation includes the work that is performed by the calling application process and the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

CONCURRENT ACCESS RESOLUTION
Specifies whether the database manager should wait for data that is in the process of being updated. DEFAULT is the default.

DEFAULT
Specifies that the concurrent access resolution is not explicitly set for this procedure. The value that is in effect when the procedure is called will be used.

WAIT FOR OUTCOME
Specifies that the database manager is to wait for the commit or rollback of data in the process of being updated.

USE CURRENTLY COMMITTED
Specifies that the database manager is to use the currently committed version of the data when encountering data that is in the process of being updated.

When the lock contention is between a read transaction and a delete or update transaction, the clause is applicable to scans with isolation level CS (but not for CS KEEP LOCKS).

WRAPPED *obfuscated-statement-text*
Specifies the encoded definition of the procedure. A CREATE PROCEDURE statement can be encoded using the WRAP scalar function.

SET OPTION-statement
Specifies the options that will be used to create the procedure. These options also apply to any default value expressions. For example, to create a debuggable procedure, the following statement could be included:

```
SET OPTION DBGVIEW = *SOURCE
```

The default values for the options depend the options in effect at create time. For more information, see "SET OPTION" on page 1368.

The options CNULIGN, CNULRQD, COMPILEOPT, NAMING, and SQLCA are not allowed in the CREATE PROCEDURE statement. The following options are used when processing default value expressions: ALWCPYDTA, CONACC, DATFMT, DATSEP, DECFLTRND, DECMPT, DECRESULT, DFTRDBCOL, LANGID, SQLCURRULE, SQLPATH, SRTSEQ, TGTRLS, TIMFMT, and TIMSEP.

SQL-routine-body
Specifies a single *SQL-procedure-statement*, including a compound statement. See Chapter 7, "SQL control statements," on page 1427 for more information about defining SQL procedures.

The SET TRANSACTION statement is not allowed in a procedure that is running on a remote application server. COMMIT and ROLLBACK statements are not allowed in an ATOMIC SQL procedure or in a procedure that is running on a connection to a remote application server.

ALTER PROCEDURE (SQL), ALTER FUNCTION (SQL Scalar), and ALTER FUNCTION (SQL Table) with a REPLACE keyword are not allowed in an *SQL-routine-body*.

Notes

General considerations for defining procedures: See "CREATE PROCEDURE" on page 937 for general information on defining procedures.

Procedure ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the procedure is created exists, the *owner* of the procedure is that user profile.
- Otherwise, the *owner* of the procedure is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the procedure is the user profile or group user profile of the job executing the statement.

Procedure authority: If SQL names are used, procedures are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, procedures are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the procedure is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the procedure.

REPLACE rules: When a procedure is recreated by REPLACE:

- Any existing comment or label is discarded.
- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved.

Error handling in procedures: Consideration should be given to possible exceptions that can occur for each SQL statement in the body of a procedure. Any exception SQLSTATE that is not handled within the procedure using a handler within a compound statement results in the exception SQLSTATE being returned to the caller of the procedure. Values of arguments passed to a procedure that correspond to OUT parameters are undefined and those that correspond to INOUT parameters are unchanged when an error is returned by the procedure.

Creating the procedure: When an SQL procedure is created, SQL creates a temporary source file that will contain C source code with embedded SQL statements. A program or service program object is then created using the CRTPGM or CRTSRVPGM command. The SQL options used to create the program are the options that are in effect at the time the CREATE PROCEDURE statement is executed. The program is created with ACTGRP(*CALLER).

When an SQL procedure is created, the procedure's attributes are stored in the created program or service program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the attributes are used to update the catalogs.

The specific procedure name is used as the name of the member in the source file and the name of the program object, if it is a valid system name. If the procedure name is not a valid system name, a unique name is generated. If a source file member with the same name already exists, the member is overlaid. If a module or a program with the same name already exists, the objects are not overlaid, and a unique name is generated. The unique names are generated according to the rules for generating system table names.

Invoking the procedure: If a DECLARE PROCEDURE statement defines a procedure with the same name as a created procedure, and a static CALL statement where the procedure name is not identified by a variable is executed from the same source program, the attributes from the DECLARE PROCEDURE statement will be used rather than the attributes from the CREATE PROCEDURE statement.

The CREATE PROCEDURE statement applies to static and dynamic CALL statements as well as to a CALL statement where the procedure name is identified by a variable.

SQL procedures must be called using the SQL CALL statement. When called, the SQL procedure runs in the activation group of the calling program.

Obfuscated statements: A CREATE PROCEDURE statement can be executed in obfuscated form. In an obfuscated statement, only the procedure name and parameters are readable followed by the WRAPPED keyword. The rest of the statement is encoded in such a way that it is not readable but can be decoded by a database server that supports obfuscated statements. Obfuscated statements can be produced by invoking the WRAP scalar function. Any debug options that are specified when the procedure is created from an obfuscated statement are ignored. A procedure that is created from an obfuscated statement cannot be restored to a release where obfuscation is not supported.

Setting of the default value: Parameters of a procedure that are defined with a default value are set to their default value when the procedure is invoked, but only if a value is not supplied for the corresponding argument, or the argument is specified as DEFAULT.

Dependent objects: An SQL routine is dependent on objects that are referenced in the *SQL-routine-body*. The names of the dependent objects are stored in catalog view SYSROUTINEDEP. If the object reference in the *SQL-routine-body* is a fully qualified name or, in SQL naming, if an unqualified name is qualified by the current schema, then the schema name of the object in SYSROUTINEDEP will be set to the specified name or the value of the current schema. Otherwise, the schema name is not set to a specific schema name. If a name is not set to a specific

schema name, then DROP and ALTER statements will not be able to determine whether the routine is dependent on the object being altered or dropped.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL can be used as synonyms for CALLED ON NULL INPUT.
- DYNAMIC RESULT SET, RESULT SETS, and RESULT SET may be used as synonyms for DYNAMIC RESULT SETS.

Example

Create an SQL procedure that returns the median staff salary. Return a result set containing the name, position, and salary of all employees who earn more than the median salary.

```
CREATE PROCEDURE MEDIAN_RESULT_SET (OUT medianSalary DECIMAL(7,2))
     LANGUAGE SQL
     DYNAMIC RESULT SETS 1
BEGIN
     DECLARE v_numRecords INTEGER DEFAULT 1;
     DECLARE v_counter INTEGER DEFAULT 0;
     DECLARE c1 CURSOR FOR
        SELECT salary
           FROM staff
           ORDER BY salary;
     DECLARE c2 CURSOR WITH RETURN FOR
        SELECT name, job, salary
           FROM staff
           WHERE salary > medianSalary
           ORDER BY salary;
     DECLARE EXIT HANDLER FOR NOT FOUND
        SET medianSalary = 6666;
     SET medianSalary = 0;
     SELECT COUNT(*) INTO v_numRecords FROM STAFF;
     OPEN c1;
     WHILE v_counter < (v_numRecords / 2 + 1)
        DO FETCH c1 INTO medianSalary;
        SET v_counter = v_counter + 1;
     END WHILE;
     CLOSE c1;
     OPEN c2;
END
```

CREATE SCHEMA

The CREATE SCHEMA statement defines a schema at the current server and optionally creates tables, views, aliases, indexes, and distinct types. Comments and labels may be added in the catalog description of tables, views, aliases, indexes, columns, and distinct types. Table, view, and distinct type privileges can be granted to users.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The *USE system authority to the following CL commands:
  - Create Library (CRTLIB)
  - If WITH DATA DICTIONARY is specified, Create Data Dictionary (CRTDTADCT)
- Administrative authority

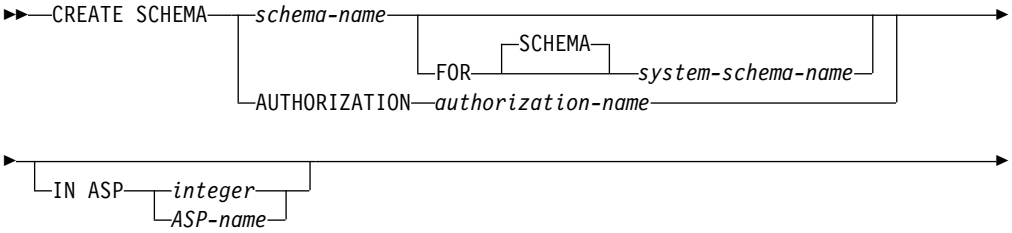
The privileges held by the authorization ID of the statement must include at least one of the following:

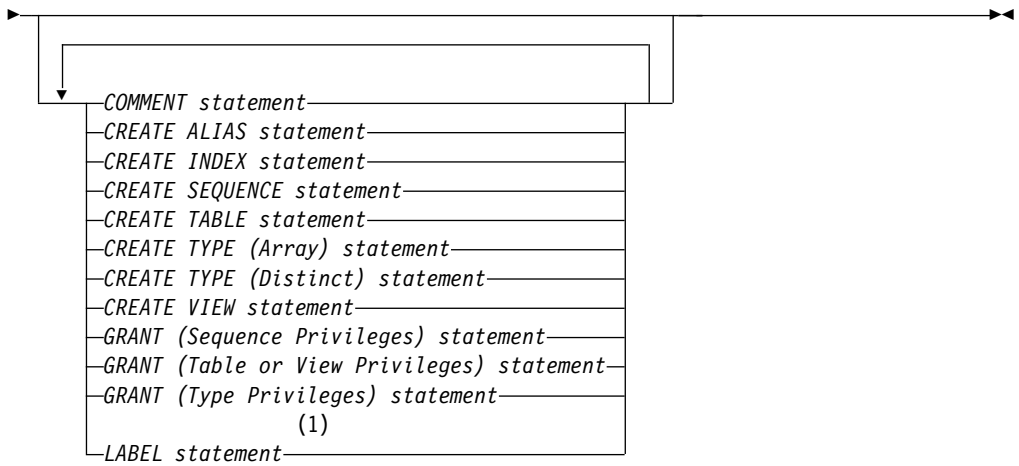
- The privileges defined for each SQL statement included in the CREATE SCHEMA statement
- Administrative authority

If the AUTHORIZATION clause is specified, the privileges held by the authorization ID of the statement must also include at least one of the following:

- The system authority *ADD to the user profile identified by authorization-name
- Administrative authority

Syntax





Notes:

1 Labels and comments on packages, procedures, functions, and parameters are not supported in the CREATE SCHEMA statement.

Description

schema-name
Names the schema. A schema is created using this name. If schema-name is specified, the authorization ID of the statement is the run-time authorization ID. The name must not be the same as the name of an existing schema at the current server. The name should not begin with 'SYS' or 'Q'. Such schema names indicate that the schema is a *system schema*.

If the *schema-name* is not a valid system name and a *system-schema-name* is not specified, SQL will generate a system name. For information about the rules for generating the name, see "Rules for Schema Name Generation" on page 972.

FOR SCHEMA *system-schema-name*
Identifies the system name of the schema. *system-schema-name* must not be the same as a schema that already exists at the current server. The *system-schema-name* must be an unqualified system identifier that is a valid system name.

If both *schema-name* and *system-schema-name* are specified, they cannot both be valid system names.

authorization-name
Identifies the authorization ID of the statement. This authorization name is also the schema-name. The name must not be the same as the name of an existing schema at the current server.

IN ASP *integer*
Specifies the auxiliary storage pool (ASP) in which to create the schema. The integer must be between 1 and 32. If 1 is specified, the schema is created on the system ASP. If this clause is omitted, an ASP of 1 is assumed.

IN ASP *ASP-name*
Specifies the auxiliary storage pool (ASP) in which to create the schema. The name must identify an auxiliary storage pool that exists at the current server.

COMMENT statement
Adds or replaces comments in the catalog descriptions of tables, views,

indexes, aliases, types, sequences, columns, or constraints. Comments on packages, functions, procedures, parameters, triggers, variables, and XSR objects are not allowed. See the COMMENT statement "COMMENT" on page 814.

CREATE ALIAS statement
Creates an alias into the schema. See the CREATE ALIAS statement "CREATE ALIAS" on page 847.

CREATE INDEX statement
Creates an index into the schema. See the CREATE INDEX statement "CREATE INDEX" on page 929.

CREATE SEQUENCE statement
Creates a sequence into the schema. See the CREATE SEQUENCE statement "CREATE SEQUENCE" on page 974.

CREATE TABLE statement
Creates a table into the schema. See the CREATE TABLE statement "CREATE TABLE" on page 982.

CREATE TYPE (Array) statement
Creates an array type into the schema. See the CREATE TYPE (Array) statement "CREATE TYPE (Array)" on page 1050.

CREATE TYPE (Distinct) statement
Creates a user-defined distinct type into the schema. See the CREATE TYPE (Distinct) statement "CREATE TYPE (Distinct)" on page 1055.

CREATE VIEW statement
Creates a view into the schema. See the CREATE VIEW statement "CREATE VIEW" on page 1069.

GRANT (Sequence Privileges) statement
Grants privileges for sequences in the schema. See the GRANT statement "GRANT (Sequence Privileges)" on page 1233.

GRANT (Table or View Privileges) statement
Grants privileges for tables and views in the schema. See the GRANT statement "GRANT (Table or View Privileges)" on page 1236.

GRANT (Type Privileges) statement
Grants privileges for types in the schema. See the GRANT statement "GRANT (Type Privileges)" on page 1242.

LABEL statement
Adds or replaces labels in the catalog descriptions of tables, views, indexes, aliases, types, sequences, columns, or constraints in the schema. Labels on packages, functions, procedures, triggers, variables, and XSR objects are not allowed. See the LABEL statement"LABEL" on page 1263.

Notes

Schema attributes: A schema is created as:

- A library: A library groups related objects, and allows you to find objects by name.
- A catalog: A catalog contains descriptions of the tables, views, indexes, and packages in the schema. A catalog consists of a set of views. For more information, see SQL Programming.

- A journal and journal receiver: A journal QSQJRN and journal receiver QSQJRN0001 is created in the schema, and is used to record changes to all tables subsequently created in the schema. For more information, see Journal Management.

An index created over a distributed table is created on all of the servers across which the table is distributed. For more information about distributed tables, see DB2 Multisystem.

Object ownership: The owner of the schema and created objects is determined as follows:

- If an AUTHORIZATION clause is specified, the specified authorization ID owns the schema and all objects created by the statement.
- Otherwise, the *owner* of the schema and all objects created by the statement is the user profile or the group user profile of the job executing the statement.

Object authority: If SQL names are used, the schema and any other objects are created with the system authority of *EXCLUDE on *PUBLIC and the library is created with the create authority parameter CRTAUT(*EXCLUDE). The owner is the only user having any authority to the schema. If other users require authority to the schema, the owner can grant authority to the objects created; using the CL command Grant Object Authority (GRTOBJAUT).

If system names are used, the schema and any other objects are created with the system authority given to *PUBLIC is determined by the system value QCRTAUT, and the library is created with CRTAUT(*SYSVAL). For more information about system security, see Security Reference, and SQL Programming.

If the owner of the schema is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the schema.

Object names: If a CREATE TABLE, CREATE INDEX, CREATE ALIAS, CREATE TYPE, CREATE SEQUENCE, or CREATE VIEW statement contains a qualified name for the table, index, alias, distinct type, sequence, or view being created, the schema name specified in that qualified name must be the same as the name of the schema being created. Any other object names referenced within the schema definition may be qualified by any schema name. Unqualified table, index, alias, distinct type, sequence, or view names in any SQL statement are implicitly qualified with the name of the created schema.

Delimiters are not used between the SQL statements.

SQL statement length: If the CREATE SCHEMA statement is executed via the RUNSQLSTM command, the maximum length of any individual CREATE TABLE, CREATE INDEX, CREATE TYPE, CREATE ALIAS, CREATE SEQUENCE, CREATE VIEW, COMMENT, LABEL, or GRANT statements within the CREATE SCHEMA statement is 2 097 152. Otherwise, the entire CREATE SCHEMA statement is limited to 2 097 152.

Name resolution performance: The name of the schema can affect the performance of statements that reference objects in the schema. If the length of a schema name is greater than 30, the performance of finding objects in the schema will be worse

than schemas whose name length is less than or equal to 30. To minimize the performance impact, ensure that the first 5 characters of the system name and the schema name are the same.

Syntax alternatives: The COLLECTION keyword can be used as a synonym for SCHEMA for compatibility to prior releases. This keyword is non-standard and should not be used.

Deprecated features: The WITH DATA DICTIONARY clause causes an IDDU data dictionary to be created in the schema. While the clause can still be specified at the end of the CREATE SCHEMA statement and is still supported; it is not recommended.

A schema created with a data dictionary cannot contain tables with LOB, XML, or DATALINK columns. The clause has no effect on the creation of catalog views.

Rules for Schema Name Generation

A system name will be generated if a schema is created with a name that is longer than 10 characters.

The SQL name or its corresponding system name may both be used in SQL statements to access the schema once it is created. However, the SQL name is only recognized by DB2 for i and the system name must be used in other environments.

If the *schema-name* is an ordinary identifier and longer than 10 characters, a 10-character *system-schema-name* will be generated as:

- The first 5 characters of the name
- A 5 digit unique number

For example:

```
The system-schema-name for LONGSCHEMANAME would be LONGS00001
```

If the *schema-name* is a delimited identifier and longer than 10 characters, a 10-character *system-schema-name* will be generated as:

- The first 4 characters from within the delimiters will be used as the first characters of the *system-schema-name*.
- If the first 4 characters are all uppercase letters, digits, or underscores, an underscore and a 5 digit unique number is appended.
- Otherwise, a 4 digit unique number is appended.

For example:

```
The system name for "longschemaname" would be "long0001"
The system name for "LONGSchemaName" would be LONG_00001
```

Examples

Example 1: Create a schema that has an inventory part table and an index over the part number. Give authority to the schema to the user profile JONES.

```
CREATE SCHEMA INVENTORY

CREATE TABLE PART (PARTNO SMALLINT NOT NULL,
                   DESCR  VARCHAR(24),
                   QUANTITY INT)
```

```
CREATE INDEX PARTIND ON PART (PARTNO)

GRANT ALL ON PART TO JONES
```

Example 2: Create a schema using the authorization ID of SMITH. Create a student table that has a comment on the student number column.

```
CREATE SCHEMA AUTHORIZATION SMITH

CREATE TABLE SMITH.STUDENT (STUDNBR SMALLINT NOT NULL UNIQUE,
                            LASTNAME CHAR(20),
                            FIRSTNAME CHAR(20),
                            ADDRESS CHAR(50))

COMMENT ON STUDENT (STUDNBR IS 'THIS IS A UNIQUE ID#')
```

CREATE SEQUENCE

The CREATE SEQUENCE statement creates a sequence at the application server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - *USE to the Create Data Area (CRTDTAARA) command
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSSEQOBJECTS catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

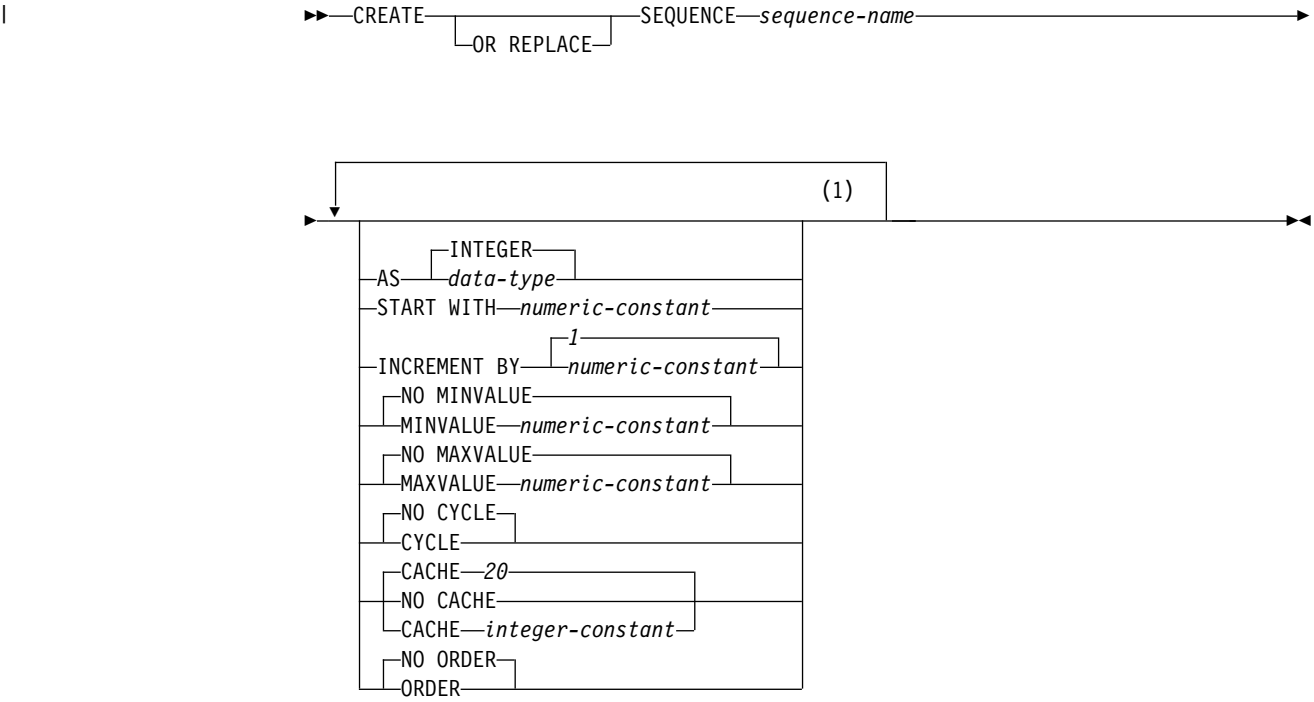
- For the distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

To replace an existing sequence, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority of *OBJMGT on the data area associated with the sequence
  - All authorities needed to DROP the sequence
  - The system authority *READ to the SYSSEQOBJECTS catalog table
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Sequence and Corresponding System Authorities When Checking Privileges to a Distinct Type.

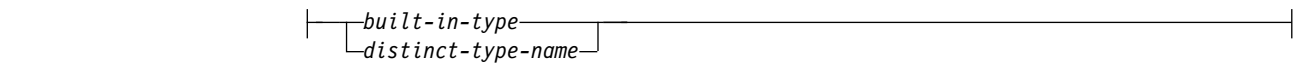
Syntax



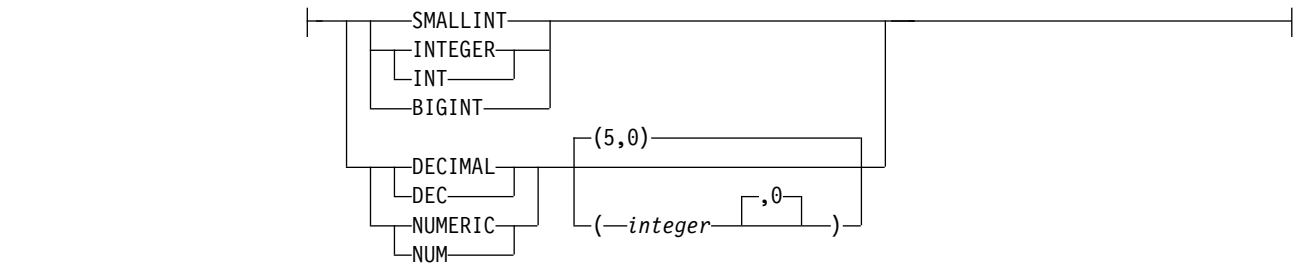
Notes:

1 The same clause must not be specified more than once.

data-type:



built-in-type:



Description

OR REPLACE
Specifies to replace the definition for the sequence if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog with the exception that privileges that were granted on the sequence are not affected. This option is ignored if a definition for the sequence does not exist at the current server.

sequence-name
Names the sequence. The name, including the implicit or explicit qualifier, must not identify a sequence or data area that already exists at the current server. If a qualified sequence name is specified, the *schema-name* cannot be QSYS2, QSYS, or SYSIBM.

If SQL names were specified, the sequence will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the sequence will be created in the schema that is specified by the qualifier. If not qualified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the sequence will be created in the current library (*CURLIB).
- Otherwise, the sequence will be created in the current schema.

AS *data-type*
Specifies the data type to be used for the sequence value. The data type can be any exact numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or NUMERIC) with a scale of zero, or a user-defined distinct type for which the source type is an exact numeric type with a scale of zero. The default is INTEGER.

built-in-type
Specifies the built-in data type used as the basis for the internal representation of the sequence. If the data type is DECIMAL or NUMERIC, the precision must be less than or equal to 63 and the scale must be 0. See "CREATE TABLE" on page 982 for a more complete description of each built-in data type.

For portability of applications across platforms, use DECIMAL instead of a NUMERIC data type.

distinct-type-name
Specifies that the data type of the sequence is a distinct type (a user-defined data type). If the source type is DECIMAL or NUMERIC, the precision of the sequence is the precision of the source type of the distinct type. The precision of the source type must be less than or equal to 63 and the scale must be 0. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path.

START WITH *numeric-constant*
Specifies the first value that is generated for the sequence. The value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence, without non-zero digits to the right of the decimal point.

If a value is not explicitly specified when the sequence is defined, the default is the MINVALUE for an ascending sequence and the MAXVALUE for a descending sequence.

This value is not necessarily the value that a sequence would cycle to after reaching the maximum or minimum value of the sequence. The START WITH clause can be used to start a sequence outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the sequence. The value can be any positive or negative value that could be assigned to a column of the

data type associated with the sequence, and does not exceed the value of a large integer constant, without nonzero digits existing to the right of the decimal point.

If the value is 0 or positive, this is an ascending sequence. If the value is negative, this is a descending sequence. The default is 1.

NO MINVALUE or MINVALUE
Specifies the minimum value at which a descending sequence either cycles or stops generating values, or an ascending sequence cycles to after reaching the maximum value. The default is NO MINVALUE.

NO MINVALUE
For an ascending sequence, the value is the START WITH value, or 1 if START WITH is not specified. For a descending sequence, the value is the minimum value of the data type (and precision, if DECIMAL or NUMERIC) associated with the sequence.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence and without non-zero digits to the right of the decimal point. The value must be less than or equal to the maximum value.

NO MAXVALUE or MAXVALUE
Specifies the maximum value at which an ascending sequence either cycles or stops generating values, or a descending sequence cycles to after reaching the minimum value. The default is NO MAXVALUE.

NO MAXVALUE
For an ascending sequence, the value is the maximum value of the data type (and precision, if DECIMAL or NUMERIC) associated with the sequence. For a descending sequence, the value is the START WITH value, or -1 if START WITH is not specified.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence and without non-zero digits to the right of the decimal point. The value must be greater than or equal to the minimum value.

NO CYCLE or CYCLE
Specifies whether this sequence should continue to generate values once the maximum or minimum value of the sequence has been reached. The default is NO CYCLE.

NO CYCLE
Specifies that values will not be generated for the sequence once the maximum or minimum value for the sequence has been reached.

CYCLE
Specifies that values continue to be generated for this sequence after the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches the maximum value of the sequence, it generates its minimum value. After a descending sequence reaches its minimum value of the sequence, it generates its maximum value. The maximum and minimum values for the column determine the range that is used for cycling.

When CYCLE is in effect, then duplicate values can be generated for the sequence.

CACHE or NO CACHE
Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of the NEXT VALUE sequence expression. The default is CACHE 20.

CACHE *integer-constant*
Specifies the maximum number of sequence values that are preallocated and kept in memory. Preallocating and storing values in the cache improves performance.

In certain situations, such as system failure, all cached sequence values that have not been used in committed statements are lost, and thus, will never be used. The value specified for the CACHE option is the maximum number of sequence values that could be lost in these situations.

The minimum value that can be specified is 2.

NO CACHE
Specifies that values for the sequence are not preallocated. If NO CACHE is specified, the performance of the NEXT VALUE sequence expression will be worse than if CACHE is specified.

ORDER or NO ORDER
Specifies whether the sequence values must be generated in order of request. The default is NO ORDER.

NO ORDER
Specifies that the sequence numbers do not need to be generated in order of request.

ORDER
Specifies that the sequence numbers are generated in order of request. If ORDER is specified, the performance of the NEXT VALUE sequence expression will be worse than if NO ORDER is specified.

Notes

Sequence attributes: A sequence is created as a *DTAARA object. The *DTAARA objects should not be changed with the Change Data Area (*CHGDTAARA) or any other similar interface because doing so may cause unexpected failures or unexpected results when attempting to use the SQL sequence through SQL.

Sequence ownership: The *owner* of the sequence is the user profile or group user profile of the job executing the statement.

Sequence authority: If SQL names are used, sequences are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, sequences are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the sequence is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the sequence.

REPLACE rules: When a sequence is recreated by REPLACE:

- Any existing comment or label is discarded.

- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved.

Relationship of MINVALUE and MAXVALUE: Typically, MINVALUE will be less than MAXVALUE, but this is not required. MINVALUE could be equal to MAXVALUE. If START WITH was the same value as MINVALUE and MAXVALUE, and CYCLE is implicitly or explicitly specified, this would be a constant sequence. In this case a request for the next value appears to have no effect because all the values generated by the sequence are in fact the same.

MINVALUE must not be greater than MAXVALUE

Defining constant sequences: It is possible to define a sequence that would always return a constant value. This could be done by specifying an INCREMENT value of zero and a START WITH value that does not exceed MAXVALUE, or by specifying the same value for START WITH, MINVALUE and MAXVALUE. For a constant sequence, each time a NEXT VALUE expression is processed the same value is returned. A constant sequence can be used as a numeric global variable. ALTER SEQUENCE can be used to adjust the values that will be generated for a constant sequence.

Defining sequences that cycle: A sequence can be cycled manually by using the ALTER SEQUENCE statement. If NO CYCLE is implicitly or explicitly specified, the sequence can be restarted or extended using the ALTER SEQUENCE statement to cause values to continue to be generated once the maximum or minimum value for the sequence has been reached.

A sequence can be explicitly defined to cycle by specifying the CYCLE keyword. Use the CYCLE option when defining a sequence to indicate that the generated values should cycle once the boundary is reached. When a sequence is defined to automatically cycle (for example CYCLE was explicitly specified), then the maximum or minimum value generated for a sequence may not be the actual MAXVALUE or MINVALUE specified, if the increment is a value other than 1 or -1. For example, the sequence defined with START WITH=1, INCREMENT=2, MAXVALUE=10 will generate a maximum value of 9, and will not generate the value 10.

When defining a sequence with CYCLE, then any application conversion tools (for converting applications from other vendor platforms to DB2) should also explicitly specify MINVALUE, MAXVALUE and START WITH.

Caching sequence numbers: A range of sequence numbers can be kept in memory for fast access. When an application accesses a sequence that can allocate the next sequence number from the cache, the sequence number allocation can happen quickly. However, if an application accesses a sequence that cannot allocate the next sequence number from the cache, the sequence number allocation will require an update to the *DTAARA object.

Choosing a high value for CACHE allows faster access to more successive sequence numbers. However, in the event of a failure, all sequence values in the cache are lost. If the NO CACHE option is used, the values of the sequence are not stored in the sequence cache. In this case every access to the sequence requires an update to the *DTAARA object. The choice of the value for CACHE should be made keeping the trade-off between performance and application requirements in mind.

Persistence of the most recently generated sequence value: The database manager remembers the most recently generated value for a sequence within the SQL-session, and returns this value for a PREVIOUS VALUE expression specifying the sequence name. The value persists until either the next value is generated for the sequence, the sequence is dropped, altered, or replaced, or until the end of the application session. The value is unaffected by COMMIT and ROLLBACK statements.

PREVIOUS VALUE is defined to have a linear scope within the application session. Therefore, in a nested application:

- on entry to a nested function, procedure, or trigger, the nested application inherits the most recently generated value for a sequence. That is, specifying an invocation of a PREVIOUS VALUE expression in a nested application will reflect sequence activity done in the invoking application, routine, or trigger before entering the nested application. An invocation of PREVIOUS VALUE expression in a nested application results in an error if a NEXT VALUE expression for the specified sequence had not yet been done in the invoking application, routine, or trigger.
- on return from a function, procedure, or trigger, the invoking application, routine or trigger will be affected by any sequence activity in the function, procedure, or trigger. That is, an invocation of PREVIOUS VALUE in the invoking application, routine, or trigger after returning from the nested application will reflect any sequence activity that occurred in the lower level applications.

Sequence journaling: When a sequence is created, journaling may be automatically started.

- If a data area called QDFTJRN exists in the same schema that the sequence is created into and the user is authorized to the data area, journaling will be started to the journal named in the data area if all the following are true:
  - The identified schema for the table must not be QSYS, QSYS2, QRECOVERY, QSPL, QRCL, QRPLOBJ, QGPL, QTEMP, SYSIBM, or any of the IASP equivalents to these libraries.
  - The journal specified in the data area must exist and the user must be authorized to start journaling to the journal.
  - The first 10 bytes of the data area must contain the name of the schema in which to find the journal.
  - The second 10 bytes must contain the name of the journal.
  - The remaining bytes contain the object types being implicitly journaled and the options that affect when implicit journaling is performed. The object type must include the value *DTAARA or *ALL. The value *NONE can be used to prevent journaling from being started.

  For more information, see the Journal Management topic collection.
- If the sequence is created into a schema that has specified (using the STRJRNLIB command) that journaling should implicitly be started.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases of other DB2 products. These keywords are non-standard and should not be used:

- The keywords NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER can be used as synonyms for NO MINVALUE, NO MAXVALUE, NO CYCLE, NO CACHE, and NO ORDER.
- A comma can be used to separate multiple sequence options.

Examples

Create a sequence called ORG_SEQ that starts at 1, increments by 1, does not cycle, and caches 24 values at a time:

```
CREATE SEQUENCE ORG_SEQ
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 24
```

The options START WITH 1, INCREMENT 1, NO MAXVALUE, and NO CYCLE are the values that would have been used if they had not been explicitly specified.

CREATE TABLE

The CREATE TABLE statement defines a table at the current server. The definition must include its name and the names and attributes of its columns. The definition may include other attributes of the table such as primary key.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - *USE to the Create Physical File (CRTPF) command
  - *CHANGE to the data dictionary if the library into which the table is created is an SQL schema with a data dictionary
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the table is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

To define a foreign key, the privileges held by the authorization ID of the statement must include at least one of the following on the parent table:

- The REFERENCES privilege or object management authority for the table
- The REFERENCES privilege on each column of the specified parent key
- Ownership of the table
- Administrative authority

If a field procedure is defined, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority *EXECUTE on the program, and
  - The system authority *EXECUTE on the library containing the program
- Administrative authority

If the LIKE clause or *select-statement* is specified, the privileges held by the authorization ID of the statement must include at least one of the following on the tables or views specified in these clauses:

- The SELECT privilege for the table or view

- Ownership of the table or view
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

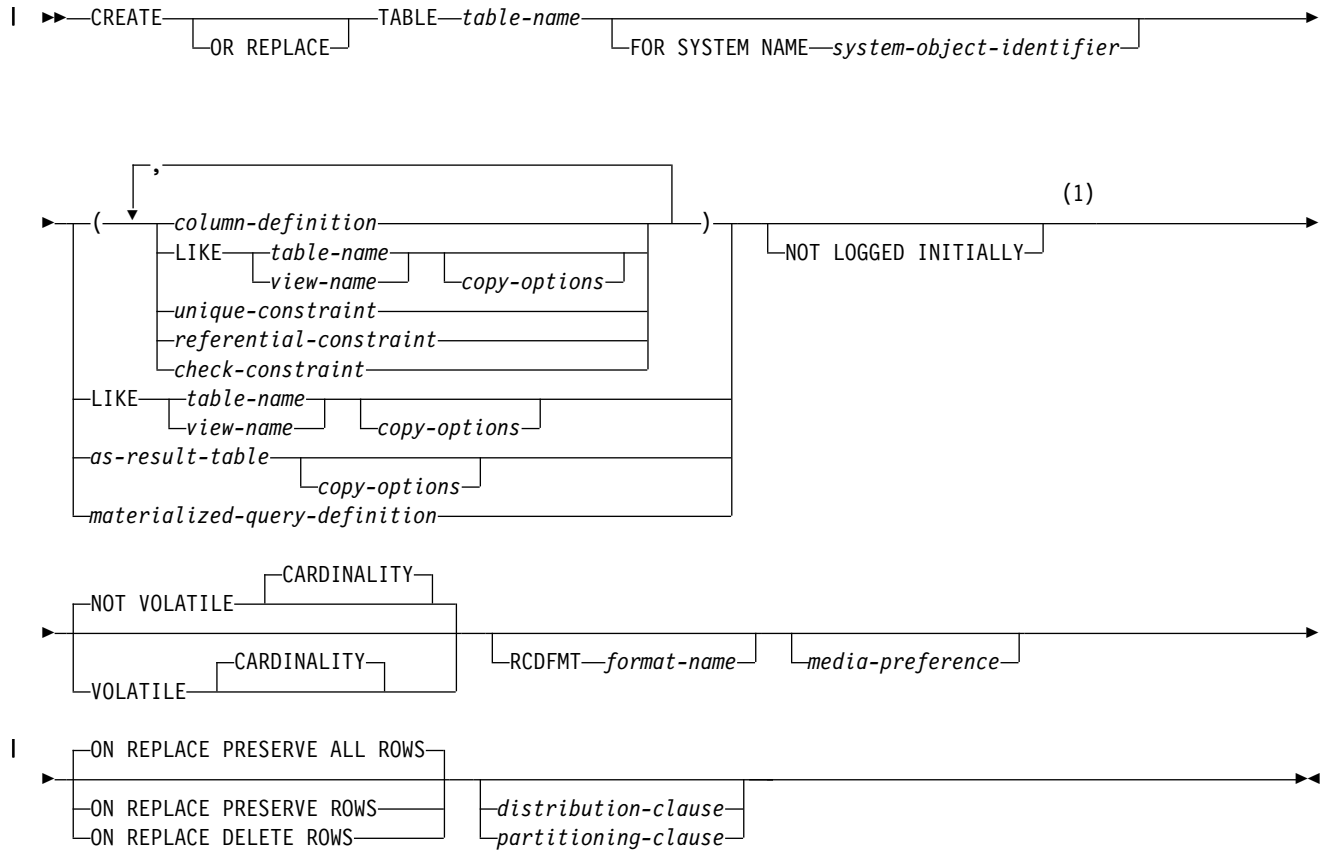
To replace an existing table, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority of *OBJMGT on the table
  - All authorities needed to DROP the table
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View and Corresponding System Authorities When Checking Privileges to a Distinct Type.

CREATE TABLE

Syntax

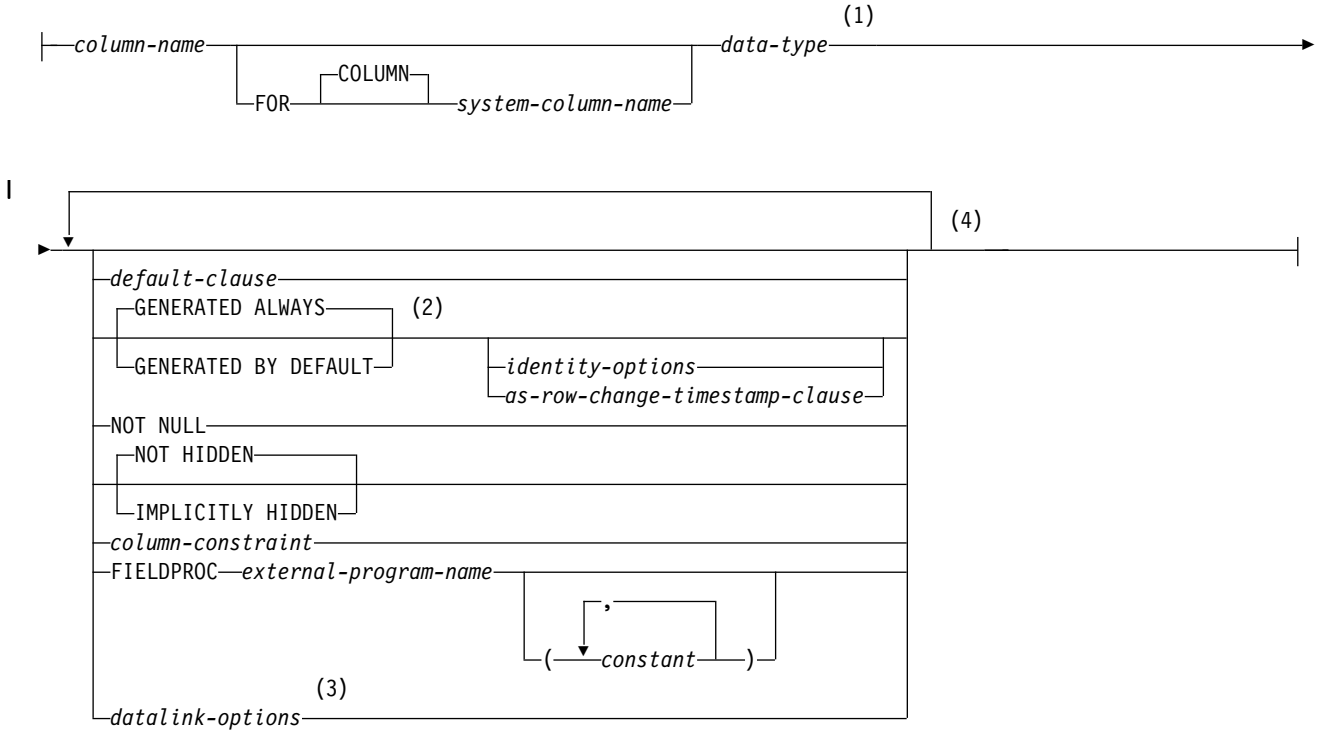


media-preference:

Notes:

1 The optional clauses can be specified in any order.

column-definition:



data-type:

Notes:

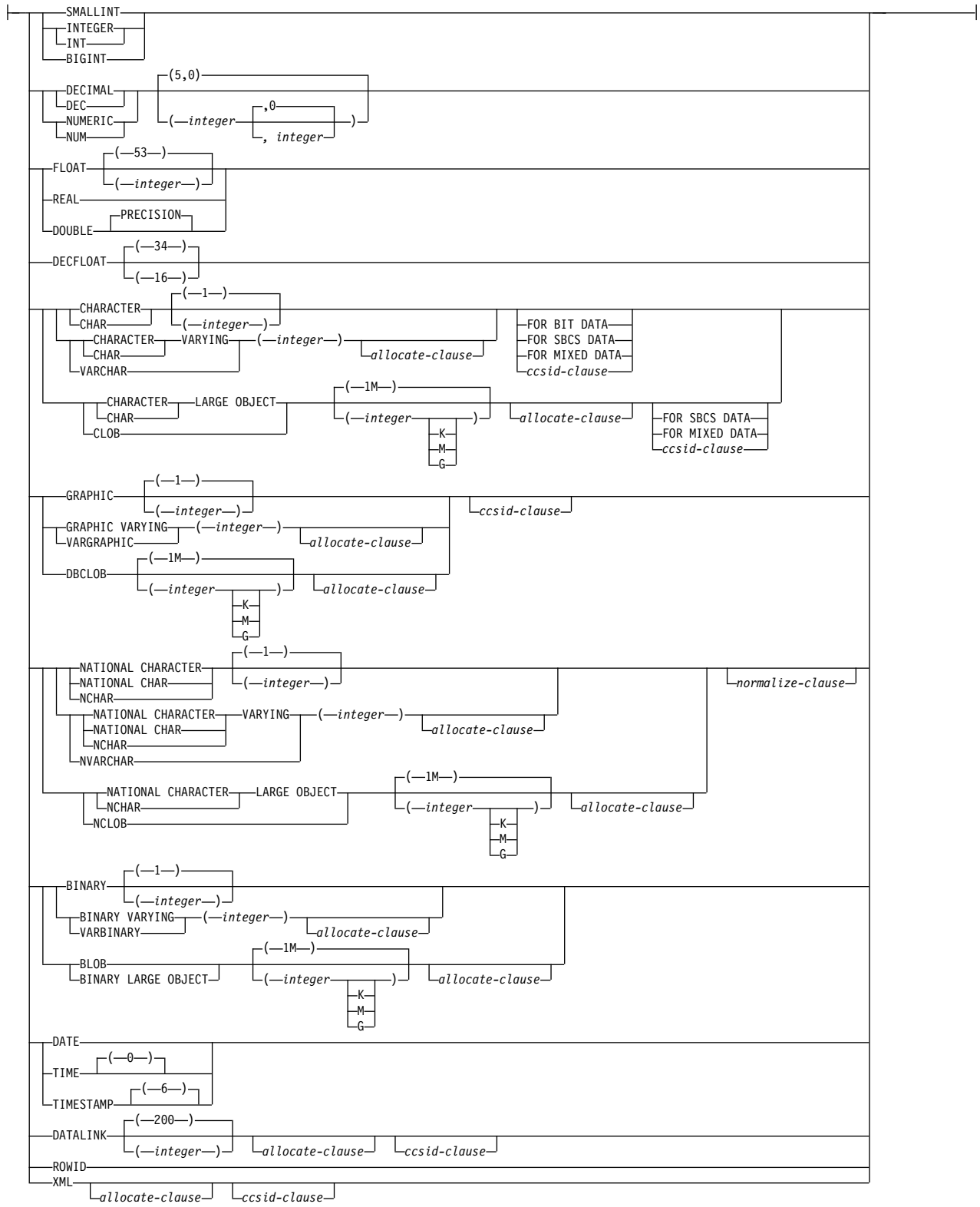
1 *data-type* is optional for row change timestamp columns

2 GENERATED can be specified only if the column has a ROWID data type (or a distinct type that is based on a ROWID data type), or the column is an identity column, or the column is a row change timestamp.

3 The *datalink-options* can only be specified for DATALINKs and distinct types sourced on DATALINKs.

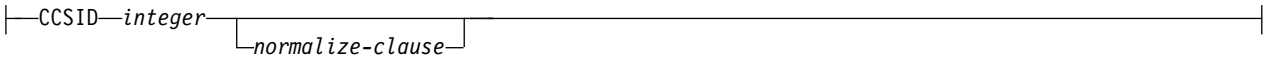
4 The same clause must not be specified more than once.

built-in-type:



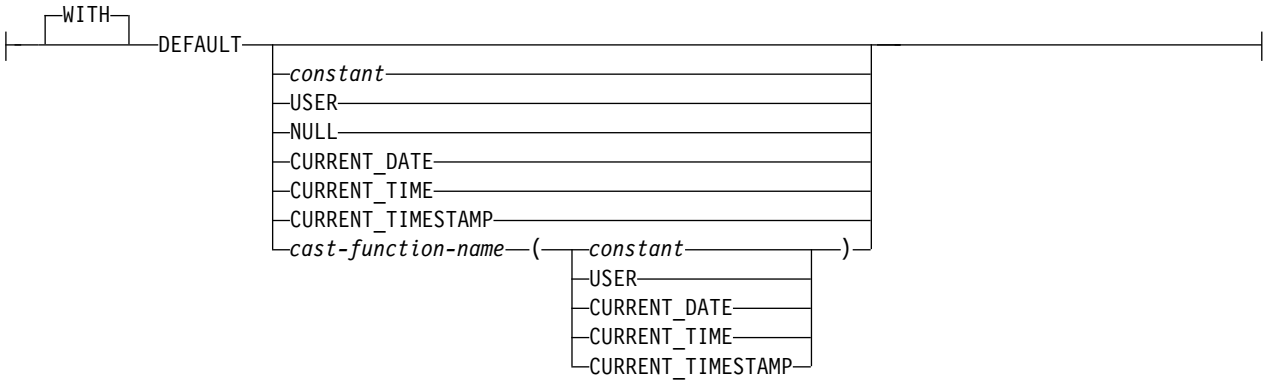
allocate-clause:

ccsid-clause:

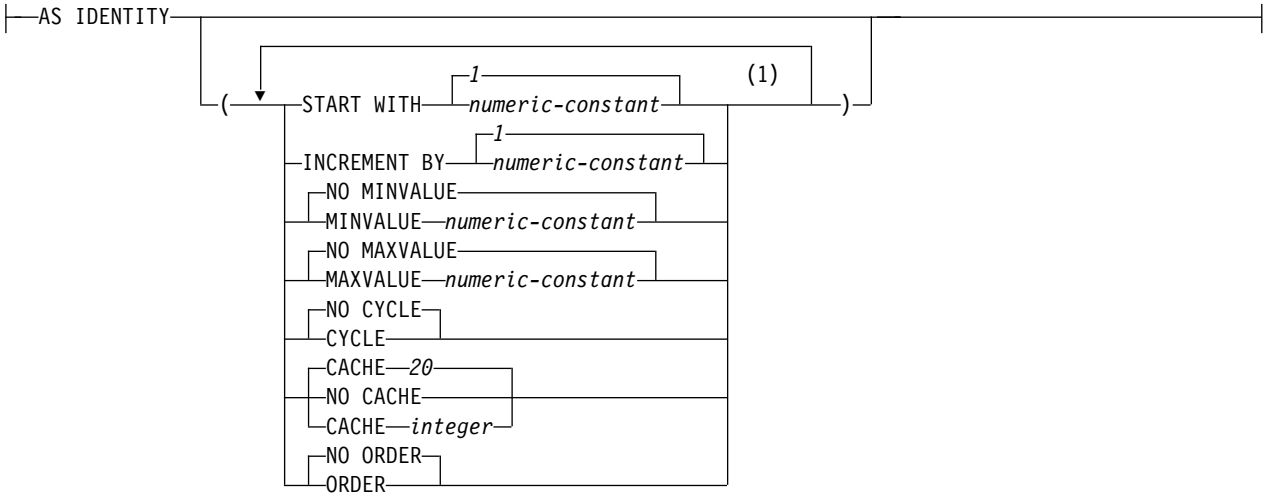


normalize-clause:

default-clause:



identity-options:

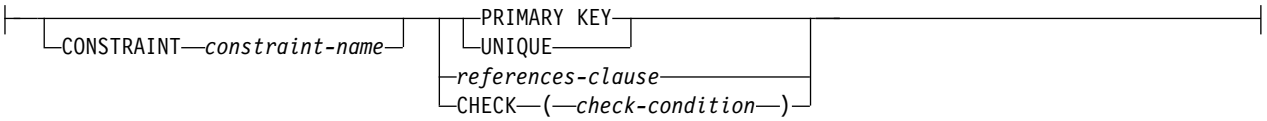


Notes:

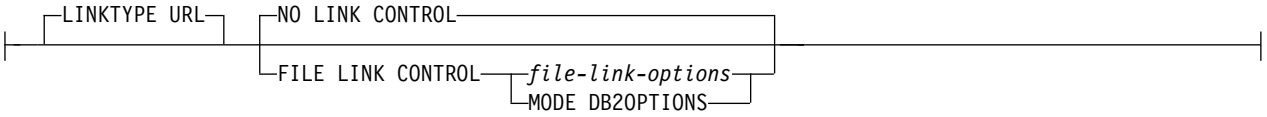
1 The same clause must not be specified more than once.

as-row-change-timestamp-clause:

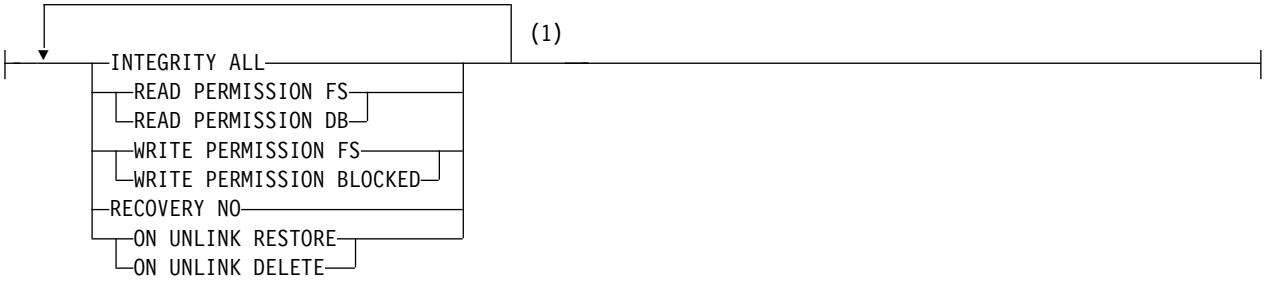
column-constraint:



datalink-options:



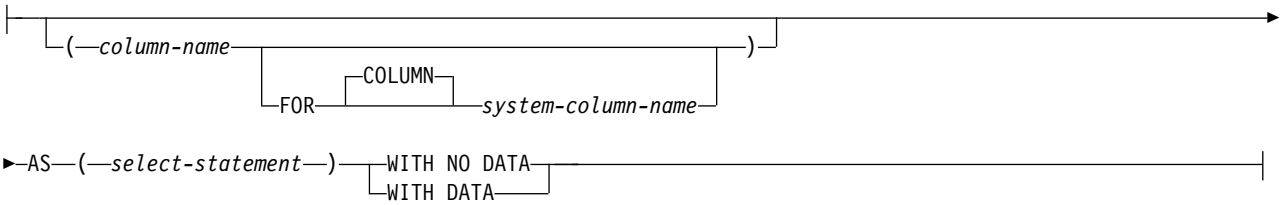
file-link-options:



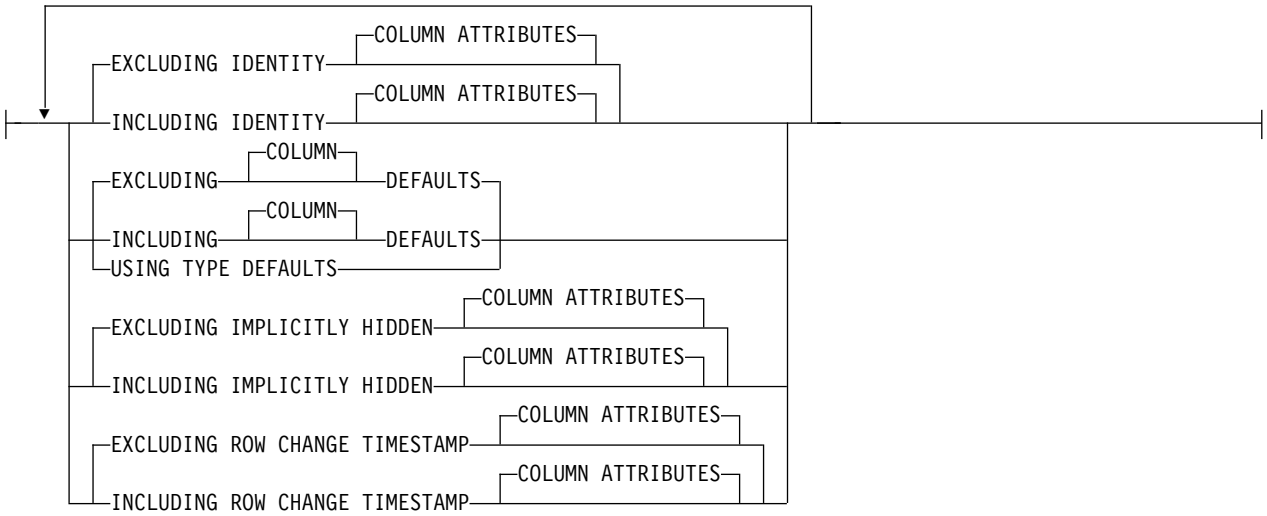
Notes:

1 All five *file-link-options* must be specified, but they can be specified in any order.

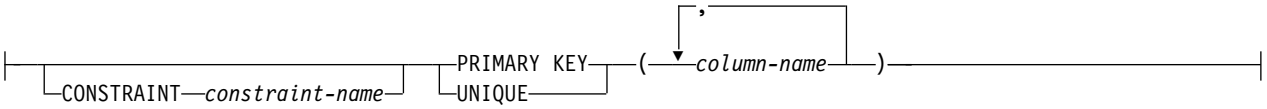
as-result-table:



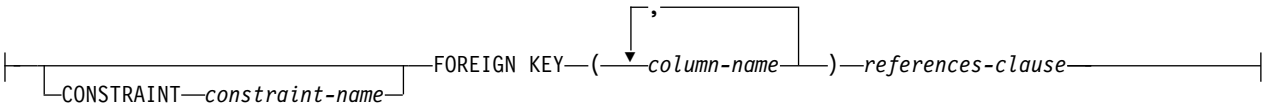
copy-options:



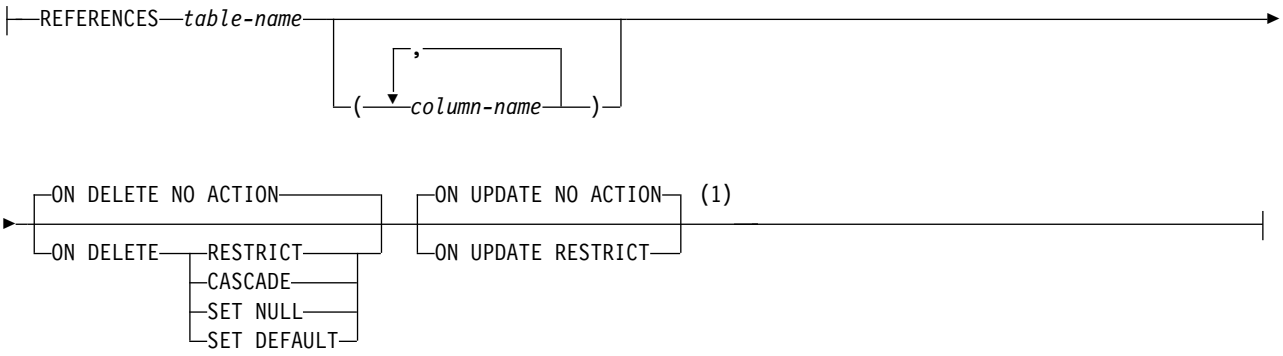
unique-constraint:



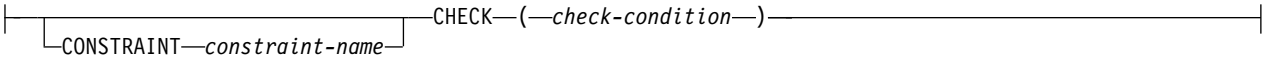
referential-constraint:



references-clause:



check-constraint:

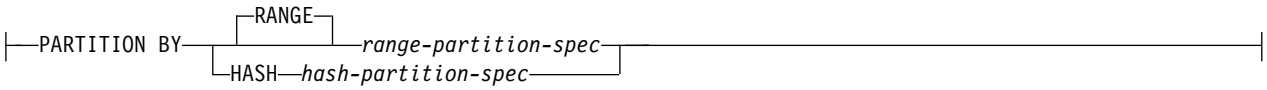


distribution-clause:

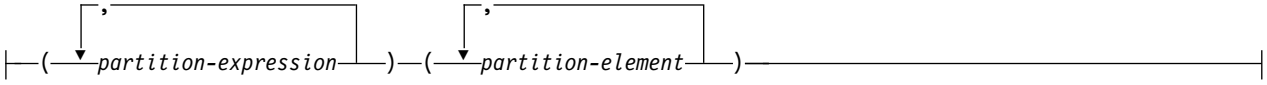
Notes:

1 The ON DELETE and ON UPDATE clauses may be specified in either order.

partitioning-clause:

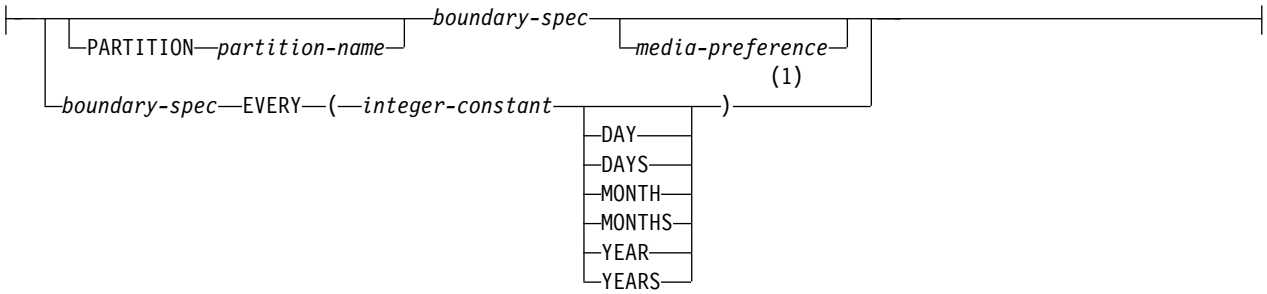


range-partition-spec:

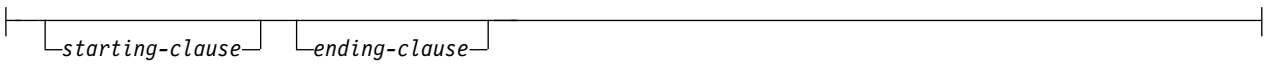


partition-expression:

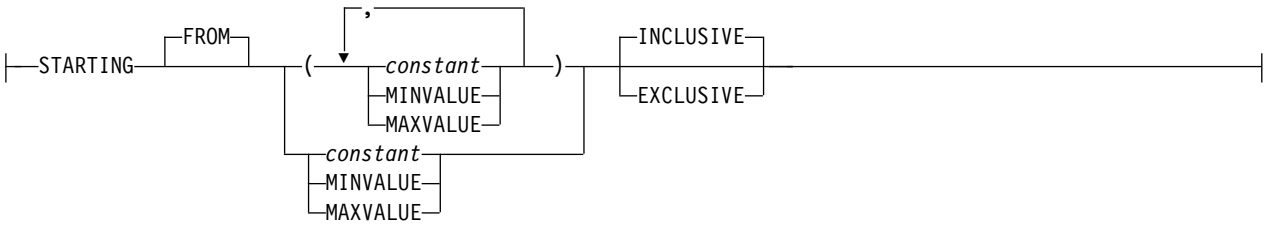
partition-element:



boundary-spec:



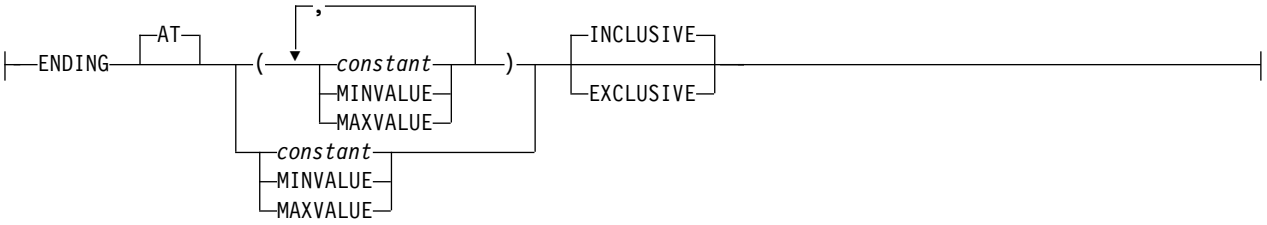
starting-clause:



Notes:

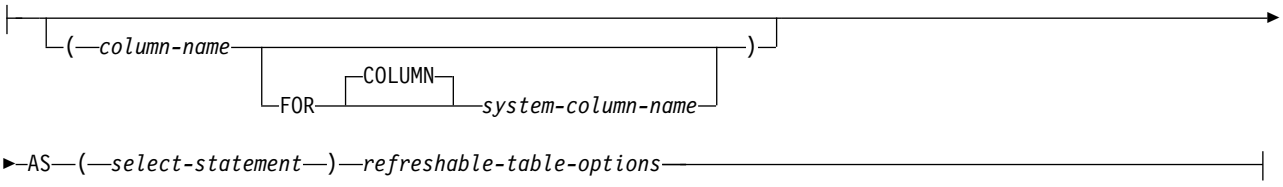
1 This syntax for a *partition-element* is valid if there is only one *partition-expression* with a numeric or datetime data type.

ending-clause:

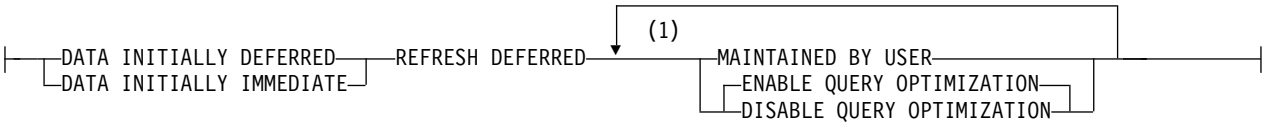


hash-partition-spec:

materialized-query-definition:



refreshable-table-options:



Notes:

1 The same clause must not be specified more than once. MAINTAINED BY USER must be specified.

Description

OR REPLACE

Specifies to replace the definition for the table if one exists at the current server. The existing definition is effectively altered before the new definition is replaced in the catalog.

A definition for the table exists if:

- FOR SYSTEM NAME is specified and the *system-object-identifier* matches the *system-object-identifier* of an existing table.
- FOR SYSTEM NAME is not specified and *table-name* is a system object name that matches the *system-object-identifier* of an existing table.

If a definition for the table exists and *table-name* is not a system object name, *table-name* can be changed to provide a new name for the table.

This option is ignored if a definition for the table does not exists at the current server.

table-name
Names the table. The name, including the implicit or explicit qualifier, must not identify an alias, file, index, table, or view that already exists at the current server.

If SQL names were specified, the table will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the table will be created in the schema that is specified by the qualifier. If not qualified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the table will be created in the current library (*CURLIB).
- Otherwise, the table will be created in the current schema.

FOR SYSTEM NAME *system-object-identifier*
Identifies the *system-object-identifier* of the table. *system-object-identifier* must not be the same as a table, view, alias, or index that already exists at the current server. The *system-object-identifier* must be an unqualified system identifier.

When *system-object-identifier* is specified, *table-name* must not be a valid system object name.

column-definition

Defines the attributes of a column. There must be at least one column definition and no more than 8000 column definitions.

The sum of the row buffer byte counts of the columns must not be greater than 32766 or, if a VARCHAR, VARGRAPHIC, or VARBINARY column is specified, 32740. Additionally, if a LOB or XML column is specified, the sum of the row data byte counts of the columns must not be greater than 3 758 096 383 at the time of insert or update. For information about the byte counts of columns according to data type, see "Maximum row sizes" on page 1027.

column-name
Names a column of the table. Do not qualify *column-name* and do not use the same name for more than one column of the table or for a *system-column-name* of the table.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see "Rules for Column Name Generation" on page 1029.

data-type
Specifies the data type of the column.

built-in-type
For *built-in-types*, use:

SMALLINT
For a small integer.

INTEGER or INT
For a large integer.

BIGINT
For a big integer.

DECIMAL(*integer***,***integer***) or DEC(***integer***,***integer***)**
DECIMAL(*integer***) or DEC(***integer***)**
DECIMAL or DEC
For a packed decimal number. The first integer is the precision of the number; that is, the total number of digits; it can range from 1 to 63. The second integer is the scale of the number (the number of digits to the right of the decimal point). It can range from 0 to the precision of the number.

You can use DECIMAL(*p*) for DECIMAL(*p*,0), and DECIMAL for DECIMAL(5,0).

NUMERIC(*integer***,***integer***) or NUM(***integer***,***integer***)**
NUMERIC(*integer***) or NUM(***integer***)**
NUMERIC or NUM
For a zoned decimal number. The first integer is the precision of the number, that is, the total number of digits; it may range from 1 to 63. The second integer is the scale of the number, (the number of digits to the right of the decimal point). It may range from 0 to the precision of the number.

You can use NUMERIC(*p*) for NUMERIC(*p*,0), and NUMERIC for NUMERIC(5,0).

FLOAT
For a double-precision floating-point number.

FLOAT(*integer***)**
For a single- or double-precision floating-point number, depending on the value of integer. The value of integer must be in the range 1 through 53. The values 1 through 24 indicate single-precision, the values 25 through 53 indicate double-precision. The default is 53.

REAL
For single-precision floating point.

DOUBLE PRECISION or DOUBLE
For double-precision floating point.

DECFLOAT(*integer***)**
DECFLOAT
For a IEEE decimal floating-point number. The value of *integer* must be either 16 or 34 and represents the number of significant digits that can be stored. If *integer* is omitted, then the DECFLOAT column will be capable of representing 34 significant digits.

CHARACTER(*integer***) or CHAR(***integer***)**
CHARACTER or CHAR
For a fixed-length character string of length *integer* bytes. The integer can range from 1 through 32766 (32765 if null capable). If FOR MIXED DATA or a mixed data CCSID is specified, the range is 4 through 32766 (32765 if null capable). If the length specification is omitted, a length of 1 is assumed.

CHARACTER VARYING (*integer***) or CHAR VARYING (***integer***) or VARCHAR (***integer***)**
For a varying-length character string of maximum length *integer* bytes, which can range from 1 through 32740 (32739 if null capable). If FOR MIXED DATA or a mixed data CCSID is specified, the range is 4 through 32740 (32739 if null capable).

CHARACTER LARGE OBJECT (*integer***[K|M|G]) or CHAR LARGE OBJECT (***integer***[K|M|G]) or CLOB (***integer***[K|M|G])**
CHARACTER LARGE OBJECT or CHAR LARGE OBJECT or CLOB
For a character large object string of the specified maximum length in bytes. The maximum length must be in the range of 1 through 2 147 483 647. If FOR MIXED DATA or a mixed data CCSID is specified, the range is 4 through 2 147 483 647. If the length specification is omitted, a length of 1 megabyte is assumed. A CLOB is not allowed in a distributed table.

integer
The maximum value for integer is 2 147 483 647. The maximum length of the string is *integer*.

integer **K**
The maximum value for integer is 2 097 152. The maximum length of the string is 1024 times *integer*.

integer **M**
The maximum value for integer is 2 048. The maximum length of the string is 1 048 576 times *integer*.

integer **G**
The maximum value for integer is 2. The maximum length of the string is 1 073 741 824 times *integer*.

GRAPHIC(*integer***)**
GRAPHIC
For a fixed-length graphic string of length *integer*, which can range from 1 through 16383 (16382 if null capable). If the length specification is omitted, a length of 1 is assumed.

VARGRAPHIC(*integer***) or GRAPHIC VARYING(***integer***)**
For a varying-length graphic string of maximum length *integer*, which can range from 1 through 16370 (16369 if null capable).

DBCLOB(*integer***[K|M|G])**
DBCLOB
For a double-byte character large object string of the specified maximum length.

The maximum length must be in the range of 1 through 1 073 741 823. If the length specification is omitted, a length of 1 megabyte is assumed. A DBCLOB is not allowed in a distributed table.

integer
The maximum value for integer is 1 073 741 823. The maximum length of the string is *integer*.

integer **K**
The maximum value for integer is 1 028 576. The maximum length of the string is 1024 times *integer*.

integer **M**
The maximum value for integer is 1 024. The maximum length of the string is 1 048 576 times *integer*.

integer **G**
The maximum value for integer is 1. The maximum length of the string is 1 073 741 824 times *integer*.

NATIONAL CHARACTER (*integer***) or NATIONAL CHAR (***integer***) or NCHAR (***integer***)**

NATIONAL CHARACTER or NATIONAL CHAR or NCHAR
For a fixed-length Unicode graphic string of length *integer*, which can range from 1 through 16383 (16382 if null capable). If the length specification is omitted, a length of 1 is assumed. The CCSID is 1200.

NATIONAL CHARACTER VARYING (*integer***) or NATIONAL CHAR VARYING (***integer***) or NCHAR VARYING (***integer***) or NVARCHAR (***integer***)**
For a varying-length Unicode graphic string of maximum length *integer*, which can range from 1 through 16370 (16369 if null capable). The CCSID is 1200.

NATIONAL CHARACTER LARGE OBJECT (*integer***[K|M|G]) or NCHAR LARGE OBJECT (***integer***[K|M|G]) or NCLOB(***integer***[K|M|G])**
NATIONAL CHARACTER LARGE OBJECT or NCHAR LARGE OBJECT or NCLOB
For a Unicode double-byte character large object string of the specified maximum length.

The maximum length must be in the range of 1 through 1 073 741 823. If the length specification is omitted, a length of 1 megabyte is assumed. The CCSID is 1200. An NCLOB is not allowed in a distributed table.

integer
The maximum value for integer is 1 073 741 823. The maximum length of the string is *integer*.

integer **K**
The maximum value for integer is 1 028 576. The maximum length of the string is 1024 times *integer*.

integer **M**
The maximum value for integer is 1 024. The maximum length of the string is 1 048 576 times *integer*.

integer **G**
The maximum value for integer is 1. The maximum length of the string is 1 073 741 824 times *integer*.

BINARY(*integer***)**
BINARY
For a fixed-length binary string of length *integer*. The integer can range from 1 through 32766 (32765 if null capable). If the length specification is omitted, a length of 1 is assumed.

BINARY VARYING (*integer***) or VARBINARY(***integer***)**
For a varying-length binary string of maximum length *integer*, which can range from 1 through 32740 (32739 if null capable).

BLOB(*integer***[K|M|G]) or BINARY LARGE OBJECT(***integer***[K|M|G])**
BLOB or BINARY LARGE OBJECT
For a binary large object string of the specified maximum length. The maximum length must be in the range of 1 through 2 147 483 647. If the length specification is omitted, a length of 1 megabyte is assumed. A BLOB is not allowed in a distributed table.

integer
The maximum value for integer is 2 147 483 647. The maximum length of the string is *integer*.

integer **K**
The maximum value for integer is 2 097 152. The maximum length of the string is 1024 times *integer*.

integer **M**
The maximum value for integer is 2 048. The maximum length of the string is 1 048 576 times *integer*.

integer **G**
The maximum value for integer is 2. The maximum length of the string is 1 073 741 824 times *integer*.

DATE
For a date.

TIME
For a time.

TIMESTAMP
For a timestamp.

DATALINK(*integer* **) or DATALINK**
For a DataLink of the specified maximum length. The maximum length must be in the range of 1 through 32717. If FOR MIXED DATA or a mixed data CCSID is specified, the range is 4 through 32717. The specified length must be sufficient to contain both the largest expected URL and any DataLink comment. If the length specification is omitted, a length of 200 is assumed. A DATALINK is not allowed in a distributed table.

A DATALINK value is an encapsulated value with a set of built-in scalar functions. The DLVALUE function creates a DATALINK value. The following functions can be used to extract attributes from a DATALINK value.

- DLCOMMENT
- DLLINKTYPE
- DLURLCOMPLETE
- DLURLPATH
- DLURLPATHONLY
- DLURLSCHEME
- DLURLSERVER

A DataLink cannot be part of any index. Therefore, it cannot be included as a column of a primary key, foreign key, or unique constraint.

ROWID
For a row ID. Only one ROWID column is allowed in a table. A ROWID is not allowed in a partitioned table.

XML
For an XML document. Only well-formed documents can be inserted into an XML column. The CCSID for the column cannot be 65535. The maximum length of the column is always 2 147 483 647 bytes.

An XML column has the following restrictions:

- The column cannot be part of any index.
- The column cannot be part of a primary, unique, or foreign key.
- The column cannot be used in a check constraint.
- A default value (WITH DEFAULT) cannot be specified for the column. If the column is nullable, the default for the column is the null value.
- The column cannot be specified in the distribution clause of a distributed table.

- The column cannot be specified in the partitioning clause of a partitioned table.

distinct-type-name
Specifies that the data type of the column is a distinct type (a user-defined data type). The length, precision, and scale of the column are respectively the length, precision, and scale of the source type of the distinct type. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path.

ALLOCATE(*integer***)**
Specifies for VARCHAR, VARGRAPHIC, VARBINARY, XML, and LOB types the space to be reserved for the column in each row. Column values with lengths less than or equal to the allocated value are stored in the fixed-length portion of the row. Column values with lengths greater than the allocated value are stored in the variable-length portion of the row and require additional input/output operations to retrieve. The allocated value may range from 1 to maximum length of the string, subject to the maximum row buffer size limit. For information about the maximum row buffer size, see "Maximum row sizes" on page 1027. If FOR MIXED DATA or a mixed data CCSID is specified, the range is 4 to the maximum length of the string. If the allocated length specification is omitted, an allocated length of 0 is assumed. For VARGRAPHIC, the integer is the number of DBCS or Unicode graphic characters. If a constant is specified for the default value and the ALLOCATE length is less than the length of the default value, the ALLOCATE length is assumed to be the length of the default value.

FOR BIT DATA
Specifies that the values of the column are not associated with a coded character set and are never converted. FOR BIT DATA is only valid for CHARACTER or VARCHAR columns. The CCSID of a FOR BIT DATA column is 65535. FOR BIT DATA is not allowed for CLOB columns.

FOR SBCS DATA
Specifies that the values of the column contain SBCS (single-byte character set) data. FOR SBCS DATA is the default for CHAR, VARCHAR, and CLOB columns if the default CCSID at the current server at the time the table is created is not DBCS-capable or if the length of the column is less than 4. FOR SBCS DATA is only valid for CHARACTER, VARCHAR, or CLOB columns. The CCSID of FOR SBCS DATA is determined by the default CCSID at the current server at the time the table is created.

FOR MIXED DATA
Specifies that the values of the column contain both SBCS data and DBCS data. FOR MIXED DATA is the default for CHAR, VARCHAR, and CLOB columns if the default CCSID at the current server at the time the table is created is DBCS-capable and the length of the column is greater than 3. Every FOR MIXED DATA column is a DBCS-open application server field. FOR MIXED DATA is only valid for CHARACTER, VARCHAR, or CLOB columns. The CCSID of FOR MIXED DATA is determined by the default CCSID at the current server at the time the table is created.

CCSID integer
Specifies that the values of the column contain data of CCSID integer. If the integer is an SBCS CCSID, the column is SBCS data. If the integer is a mixed data CCSID, the column is mixed data and the length of the column must be greater than 3. For character columns, the CCSID must be an SBCS CCSID or a mixed data CCSID. For graphic columns, the CCSID must be a

DBCS, UTF-16, or UCS-2 CCSID. If a CCSID is not specified for a graphic column, the CCSID is determined by the default CCSID at the current server at the time the table is created. For XML columns, the CCSID must not be 65535. If a CCSID is not specified for an XML column, the CCSID is established at the time the CREATE TABLE is executed according to the SQL_XML_DATA_CCSID QAQQINI option setting. The default CCSID is 1208. See "XML Values" on page 88 for a description of this option.For a list of valid CCSIDs, see Appendix E, "CCSID values," on page 1541.

CCSID 1208 (UTF-8) or 1200 (UTF-16) data can contain *combining characters*. Combining character support allows a resulting character to be comprised of more than one character. After the first character, up to 300 different non-spacing accent characters (umlauts, accent, etc.) can follow in the data string. If the resulting character is one that is already defined in the character set, that character has more than one representation. *Normalization* replaces the string of combining characters with the hex value of the defined character. This ensures that the same character is represented in a single consistent way. If normalization is not performed, two strings that look identical will not compare equal.

NOT NORMALIZED
The data should not be normalized when passed from the application.

NORMALIZED
The data should be normalized when passed from the application.

DEFAULT
Specifies a default value for the column. This clause cannot be specified more than once in a *column-definition*. DEFAULT cannot be specified for an XML column, a ROWID column, an identity column (a column that is defined AS IDENTITY), or a row change timestamp column. The database manager generates default values for ROWID columns, identity columns, and row change timestamp columns. For an XML column, the default is NULL unless NOT NULL is specified; in that case there is no default. If a value is not specified following the DEFAULT keyword, then:

- if the column is nullable, the default value is the null value.
- if the column is not nullable, the default depends on the data type of the column:

Data type	Default value
Numeric	0
Fixed-length character or graphic string	Blanks
Fixed-length binary string	Hexadecimal zeros
Varying-length string	A string length of 0
Date	The current date at the time of INSERT
Time	The current time at the time of INSERT
Timestamp	The current timestamp at the time of INSERT
Datalink	A value corresponding to DLVALUE('','URL','')
distinct-type	The default value of the corresponding source type of the distinct type.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

constant
Specifies the constant as the default for the column. The specified constant must represent a value that could be assigned to the column in accordance with the rules of assignment as described in "Assignments and comparisons" on page 98. A floating-point constant or decimal floating-point constant must not be used for a SMALLINT, INTEGER, DECIMAL, or NUMERIC column. A decimal constant must not contain more digits to the right of the decimal point than the specified scale of the column.

USER
Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value of the column. The data type of the column must be CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the USER special register.

NULL
Specifies null as the default for the column. If NOT NULL is specified, DEFAULT NULL must not be specified within the same *column-definition*.

NULL is the only default value allowed for a datalink column.

CURRENT_DATE
Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the column must be DATE or a distinct type based on a DATE.

CURRENT_TIME
Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the column must be TIME or a distinct type based on a TIME.

CURRENT_TIMESTAMP
Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the column must be TIMESTAMP or a distinct type based on a TIMESTAMP.

cast-function-name
This form of a default value can only be used with columns defined as a distinct type, BINARY, VARBINARY, BLOB, CLOB, DBCLOB, DATE, TIME or TIMESTAMP data types. The following table describes the allowed uses of these *cast-functions*.

Data Type	Cast Function Name
Distinct type N based on a BINARY, VARBINARY, BLOB, CLOB, or DBCLOB	BINARY, VARBINARY, BLOB, CLOB, or DBCLOB *
Distinct type N based on a DATE, TIME, or TIMESTAMP	N (the user-defined cast function that was generated when N was created) ** or DATE, TIME, or TIMESTAMP *
Distinct type N based on other data types	N (the user-defined cast function that was generated when N was created) **
BINARY, VARBINARY, BLOB, CLOB, or DBCLOB	BINARY, VARBINARY, BLOB, CLOB, or DBCLOB *
DATE, TIME, or TIMESTAMP	DATE, TIME, or TIMESTAMP *

Data Type	Cast Function Name
Notes: * The name of the function must match the name of the data type (or the source type of the distinct type) with an implicit or explicit schema name of QSYS2. ** The name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type.	

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. For BINARY, VARBINARY, BLOB, CLOB, DBCLOB, DATE, TIME, and TIMESTAMP functions, the constant must be a string constant.

USER
Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value for the column. The data type of the source type of the distinct type of the column must be CHAR or VARCHAR with a length attribute greater than or equal to the length attribute of the USER special register.

CURRENT_DATE
Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the source type of the distinct type of the column must be DATE.

CURRENT_TIME
Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the source type of the distinct type of the column must be TIME.

CURRENT_TIMESTAMP
Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the source type of the distinct type of the column must be TIMESTAMP.

If the value specified is not valid, an error is returned.

GENERATED
Specifies that the database manager generates values for the column. GENERATED may be specified if the column is to be considered an identity column (defined with the AS IDENTITY clause) or a row change timestamp column. It may also be specified if the data type of the column is a ROWID (or a distinct type that is based on a ROWID). Otherwise, it must not be specified.

ALWAYS
Specifies that the database manager will always generate a value for the column when a row is inserted or updated and a default value must be generated. ALWAYS is the recommended value.

BY DEFAULT
Specifies that the database manager will generate a value for the column when a row is inserted or updated and a default value must be generated, unless an explicit value is specified.

For a ROWID column, the database manager uses a specified value, but it must be a valid unique row ID value that was previously generated by the database manager or DB2 for i.

For an identity column or row change timestamp column, the database manager inserts or updates a specified value but does not verify that it is a unique value for the column unless the identity column or row change timestamp column has a unique constraint or a unique index that solely specifies the identity column or row change timestamp column.

AS IDENTITY
Specifies that the column is an identity column for the table. A table can have only one identity column. An identity column is not allowed in a distributed table. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL, or NUMERIC with a scale of zero, or a distinct type based on one of these data types). If a DECIMAL or NUMERIC data type is specified, the precision must not be greater than 31.

An identity column is implicitly NOT NULL.

START WITH *numeric-constant*
Specifies the first value that is generated for the identity column. The value can be any positive or negative value that could be assigned to the column without non-zero digits existing to the right of the decimal point.

If a value is not explicitly specified when the identity column is defined, the default is the MINVALUE for an ascending sequence and the MAXVALUE for a descending sequence. This value is not necessarily the value that a sequence would cycle to after reaching the maximum or minimum value of the sequence. The START WITH clause can be used to start a sequence outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the identity column. The value must not exceed the value of a large integer constant without any non-zero digits existing to the right of the decimal point. The value must be assignable to the column. The default is 1.

If the value is zero or positive, the sequence of values for the identity column ascends. If the value is negative, the sequence of values descends.

MAXVALUE or MINVALUE
Specifies the maximum value at which an ascending identity column either cycles or stops generating values, or a descending identity column cycles to after reaching the minimum value.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column, but the value must be greater than the minimum value.

If a value is not explicitly specified when the identity column is defined, this is the maximum value of the data type for an ascending sequence; or the START WITH value, or -1 if START WITH was not specified, for a descending sequence.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value that is

generated for this identity column. This value can be any positive or negative value that could be assigned to this column, but the value must be less than the maximum value.

If a value is not explicitly specified when the identity column is defined, this is the START WITH value, or 1 if START WITH was not specified, for an ascending sequence; or the minimum value of the data type (and precision, if DECIMAL) for a descending sequence.

CACHE or NO CACHE
Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of inserting rows into a table.

CACHE *integer*
Specifies the number of values of the identity column sequence that the database manager preallocates and keeps in memory. The minimum value that can be specified is 2, and the maximum is the largest value that can be represented as an integer. The default is 20.

In certain situations, such as system failure, all cached identity column values that have not been used in committed statements are lost, and thus, will never be used. The value specified for the CACHE option is the maximum number of identity column values that could be lost in these situations.

NO CACHE
Specifies that values for the identity column are not preallocated.

CYCLE or NO CYCLE
Specifies whether this identity column should continue to generate values after reaching either the maximum or minimum value of the sequence.

CYCLE
Specifies that values continue to be generated for this column after the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches the maximum value of the sequence, it generates its minimum value. After a descending sequence reaches its minimum value of the sequence, it generates its maximum value. The maximum and minimum values for the column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by the database manager for an identity column. If a unique constraint or unique index exists on the identity column, and a non-unique value is generated for it, an error occurs.

NO CYCLE
Specifies that values will not be generated for the identity column once the maximum or minimum value for the sequence has been reached. This is the default.

ORDER or NO ORDER
Specifies whether the identity values must be generated in order of request.

ORDER
Specifies that the values are generated in order of request.

NO ORDER
Specifies that the values do not need to be generated in order of request. This is the default.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP
Specifies that the column is a timestamp and the values will be generated by the database manager. The database manager generates a value for the column for each row as a row is inserted, and for every row in which any column is updated. The value generated for a row change timestamp column is a timestamp corresponding to the time of the insert or update of the row. If multiple rows are inserted with a single SQL statement, the value for the row change timestamp column may be different for each row to reflect when each row was inserted. The generated value is not guaranteed to be unique.

A table can have only one row change timestamp column. If *data-type* is specified, it must be a TIMESTAMP or a distinct type based on a TIMESTAMP. A row change timestamp column cannot have a DEFAULT clause and must be NOT NULL.

NOT NULL
Prevents the column from containing null values. Omission of NOT NULL implies that the column can be null. NOT NULL is required for a row change timestamp column.

NOT HIDDEN
Indicates the column is included in implicit references to the table in SQL statements. This is the default.

IMPLICITLY HIDDEN
Indicates the column is not visible in SQL statements unless it is referred to explicitly by name. For example, SELECT * does not include any hidden columns in the result. A table must contain at least one column that is not IMPLICITLY HIDDEN.

column-constraint

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that was previously specified in the CREATE TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

PRIMARY KEY
Provides a shorthand method of defining a primary key composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

This clause must not be specified in more than one column definition and must not be specified at all if the UNIQUE clause is specified in the column definition. The column must not be a LOB, DATALINK, or XML column. If a sort sequence is specified, the column must not contain a field procedure.

When a primary key is added, a CHECK constraint is implicitly added to enforce the rule that the NULL value is not allowed in the column that makes up the primary key.

UNIQUE
Provides a shorthand method of defining a unique constraint composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE (C) clause is specified as a separate clause.

This clause cannot be specified more than once in a column definition and must not be specified if PRIMARY KEY is specified in the column definition. The column must not be a LOB, DATALINK, or XML column. If a sort sequence is specified, the column must not contain a field procedure.

references-clause
The *references-clause* of a *column-definition* provides a shorthand method of defining a foreign key composed of a single column. Thus, if a references-clause is specified in the definition of column C, the effect is the same as if that references-clause were specified as part of a FOREIGN KEY clause in which C is the only identified column. The *references-clause* is not allowed if the table is a declared global temporary table or a distributed table. The column cannot be a row change timestamp column.

CHECK(*check-condition***)**
The CHECK(*check-condition*) of a *column-definition* provides a shorthand method of defining a check constraint whose *check-condition* only references a single column. Thus, if CHECK is specified in the column definition of column C, no columns other than C can be referenced in the *check-condition* of the check constraint. The effect is the same as if the check constraint were specified as a separate clause.

ROWID, XML, and DATALINK with FILE LINK CONTROL columns cannot be referenced in a CHECK constraint. For additional restrictions see, "check-constraint" on page 1014.

FIELDPROC
Designates an *external-program-name* as the field procedure exit routine for the column. It must be an ILE program that does not contain SQL. It cannot be a service program.

The field procedure encodes and decodes column values. Before a value is inserted in the column, it is passed to the field procedure for encoding. Before a value from the column is used, it is passed to the field procedure for decoding.

The field procedure is also invoked during the processing of the CREATE TABLE statement. When so invoked, the procedure provides DB2 with the column's field description. The field description defines the data characteristics of the encoded values. By contrast, the information supplied for the column in the CREATE TABLE statement defines the data characteristics of the decoded values.

constant
Specifies a parameter that is passed to the field procedure when it is invoked. A parameter list is optional.

A field procedure cannot be defined for a column that is a ROWID or DATALINK or a distinct type based on a ROWID or DATALINK. The column must not be an identity column or a row change timestamp column. The column must not have a default value of CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP, or USER. The nullability attribute of the encoded and decoded form of the field must match. The column cannot be referenced in a check condition, unless it is referenced in a NULL predicate. If it is part of a foreign key, the corresponding parent key column must use the same field procedure. See SQL Programming for more details on how to create a field procedure.

datalink-options
Specifies the options associated with a DATALINK data type.

LINKTYPE URL
Defines the type of link as a Uniform Resource Locator (URL).

NO LINK CONTROL
Specifies that there will not be any check made to determine that the linked files exist. Only the syntax of the URL will be checked. There is no database manager control over the linked files.

FILE LINK CONTROL
Specifies that a check should be made for the existence of the linked files. Additional options may be used to give the database manager further control over the linked files.

If FILE LINK CONTROL is specified, each file can only be linked once. That is, its URL can only be specified in a single FILE LINK CONTROL column in a single table.

file-link-options
Additional options to define the level of database manager control of the linked files.

INTEGRITY
Specifies the level of integrity of the link between a DATALINK value and the actual file.

ALL
Any file specified as a DATALINK value is under the control of the database manager and may NOT be deleted or renamed using standard file system programming interfaces.

READ PERMISSION
Specifies how permission to read the file specified in a DATALINK value is determined.

FS The read access permission is determined by the file system permissions. Such files can be accessed without retrieving the file name from the column.

DB The read access permission is determined by the database. Access to the file will only be allowed by passing a valid file access token, returned on retrieval of the DATALINK value from the table, in the open operation. If READ PERMISSION DB is specified, WRITE PERMISSION BLOCKED must be specified.

WRITE PERMISSION
Specifies how permission to write to the file specified in a DATALINK value is determined.

FS The write access permission is determined by the file system permissions. Such files can be accessed without retrieving the file name from the column.

BLOCKED
Write access is blocked. The file cannot be directly updated through any interface. An alternative mechanism must be used to perform updates to the information. For example, the file is copied, the copy updated, and then the DATALINK value updated to point to the new copy of the file.

RECOVERY
Specifies whether the database manager will support point in time recovery of files referenced by values in this column.

NO Specifies that point in time recovery will not be supported.

ON UNLINK
Specifies the action taken on a file when a DATALINK value is changed or deleted (unlinked). Note that this is not applicable when WRITE PERMISSION FS is used.

RESTORE
Specifies that when a file is unlinked, the DataLink File Manager will attempt to return the file to the owner with the permissions that existed at the time the file was linked. In the case where the user is no longer registered with the file server, the result depends on the file system that contains the files. If the files are in the AIX® file system, the owner is "dfmunknown". If the files are in IFS, the owner is QDLFM. This can only be specified when INTEGRITY ALL and WRITE PERMISSION BLOCKED are also specified.

DELETE
Specifies that the file will be deleted when it is unlinked. This can only be specified when READ PERMISSION DB and WRITE PERMISSION BLOCKED are also specified.

MODE DB2OPTIONS
This mode defines a set of default file link options. The defaults defined by DB2OPTIONS are:

- INTEGRITY ALL
- READ PERMISSION FS
- WRITE PERMISSION FS
- RECOVERY NO

LIKE

table-name **or** *view-name*
Specifies that the columns of the table have exactly the same name and description as the columns of the identified table (*table-name*) or view (*view-name*). The name must identify a table or view that exists at the current server.

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table or view. The implicit definition includes the following attributes of the *n* columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID
- FIELDPROC (only copied for *table-name*)

If the LIKE clause is specified immediately following the *table-name* and not enclosed in parenthesis, the following column attributes are also included, otherwise they are not included (the default value, identity, row change timestamp, and hidden attributes can also be controlled by using the *copy-options*):

- Default value, if a *table-name* is specified (*view-name* is not specified)
- Nullability
- Hidden attributes
- Identity attributes
- Column heading and text (see “LABEL” on page 1263)

Any REFFLD information for the column will be copied to the new column definition.

The implicit definition does not include any other optional attributes of the identified table or view. For example, the new table does not automatically include primary keys, foreign keys, or triggers. The new table has these and other optional attributes only if the optional clauses are explicitly specified.

If the specified table or view is a non-SQL created physical file or logical file, any non-SQL attributes are removed. For example, the date and time format will be changed to ISO.

copy-options

INCLUDING IDENTITY COLUMN ATTRIBUTES or EXCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies whether identity column attributes are inherited.

INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the table inherits the identity attribute, if any, of the columns resulting from *select-statement*, *table-name*, or *view-name*. In general, the identity attribute is copied if the element of the corresponding column in the table, view, or *select-statement* is the name of a table column or the name of a view column that directly or indirectly maps to the name of a base table column with the identity attribute. If the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified with the AS *select-statement* clause, the columns of the new table do not inherit the identity attribute in the following cases:

- The select list of the *select-statement* includes multiple instances of an identity column name (that is, selecting the same column more than once).
- The select list of the *select-statement* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *select-statement* includes a set operation (UNION or INTERSECT).

If INCLUDING IDENTITY is not specified, the table will not have an identity column.

EXCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the table does not inherit the identity attribute, if any, of the columns resulting from the *fullselect*, *table-name*, or *view-name*.

EXCLUDING COLUMN DEFAULTS or INCLUDING COLUMN DEFAULTS or USING TYPE DEFAULTS

Specifies whether column defaults are inherited.

EXCLUDING COLUMN DEFAULTS

Specifies that the column defaults are not inherited from the definition of the source table. The default values of the column of the new table are either null or there are no default values. If the column can be null, the default is the null value. If the column cannot be null, there is no default value, and an error occurs if a value is not provided for a column on INSERT for the new table.

INCLUDING COLUMN DEFAULTS

Specifies that the table inherits the default values of the columns resulting from the *select-statement*, *table-name*, or *view-name*. A default value is the value assigned to a column when a value is not specified on an INSERT.

Do not specify INCLUDING COLUMN DEFAULTS, if you specify USING TYPE DEFAULTS.

If INCLUDING COLUMN DEFAULTS is not specified, the default values are not inherited.

USING TYPE DEFAULTS
Specifies that the default values for the table depend on the data type of the columns that result from the *select-statement*, *table-name*, or *view-name*. If the column is nullable, then the default value is the null value. Otherwise, the default value is as follows:

Data type	Default value
Numeric	0
Fixed-length character or graphic string	Blanks
Fixed-length binary string	Hexadecimal zeros
Varying-length string	A string length of 0
Date	The current date at the time of INSERT
Time	The current time at the time of INSERT
Timestamp	The current timestamp at the time of INSERT
Datalink	A value corresponding to DLVALUE('','URL','')
XML	There is no default value
distinct-type	The default value of the corresponding source type of the distinct type.

Do not specify USING TYPE DEFAULTS if INCLUDING COLUMN DEFAULTS is specified.

INCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES or EXCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES
Specifies whether implicitly hidden columns are inherited.

INCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES
Specifies that the table inherits implicitly hidden columns from *select-statement*, *table-name*, or *view-name* and those columns will be defined with the implicitly hidden attribute in the new table.

If INCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES is not specified, the table will not have any implicitly hidden columns.

EXCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES
Specifies that the table does not inherit implicitly hidden columns from the *fullselect*, *table-name*, or *view-name*.

INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES or EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies whether the row change timestamp attribute is inherited.

INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies that the table inherits the row change timestamp attribute, if any, of the columns resulting from *select-statement*, *table-name*, or *view-name*. In general, the row change timestamp attribute is copied if the element of the corresponding column in the table, view, or *select-statement* is the name of a table column or the name of a view column that directly or indirectly maps to the name of a base table column with the row change timestamp attribute. If the INCLUDING ROW CHANGE TIMESTAMP COLUMN

ATTRIBUTES clause is specified with the AS *select-statement* clause, the columns of the new table do not inherit the row change timestamp in the following cases:

- The select list of the *select-statement* includes multiple instances of a row change timestamp column name (that is, selecting the same column more than once).
- The select list of the *select-statement* includes multiple row change timestamp columns (that is, it involves a join).
- The row change timestamp column is included in an expression in the select list.
- The *select-statement* includes a set operation (UNION or INTERSECT).

If INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES is not specified, the table will not have a row change timestamp column.

EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies that the table does not inherit the row change timestamp attribute, if any, of the columns resulting from the *fullselect*, *table-name*, or *view-name*.

as-result-table

column-name
Names a column in the table. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the *select-statement*. Each *column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the table inherit the names of the columns of the result table of the *select-statement*.

A list of column names must be specified if the result table of the *select-statement* has duplicate column names or an unnamed column. An unnamed column is a column derived from a constant, function, expression, or set operation (UNION or INTERSECT) that is not named using the AS clause of the select list.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the *system-column-name* is not specified, and the column-name is not a valid *system-column-name*, a system column name is generated. For more information about how system column names are generated, see "Rules for Table Name Generation" on page 1030.

select-statement
Specifies that the columns of the table have the same name and description as the columns that would appear in the derived result table of the *select-statement* if the *select-statement* were to be executed. The use of AS *(select-statement)* is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *select-statement*.

The implicit definition includes the following attributes of the n columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID
- Nullability
- FIELDPROC

- Column heading and text (see "LABEL" on page 1263)

The following attributes are not included (they may be included by using the *copy-options*):

- Default value
- Hidden attribute
- Identity attributes
- Row change timestamp attribute

The implicit definition does not include any other optional attributes of the identified table or view. For example, the new table does not automatically include a primary key or foreign key from a table. The new table has these and other optional attributes only if the optional clauses are explicitly specified.

Any column in the *select-clause* that is either a direct reference to a column in another table or view or uses only a CAST to change the result attributes will have REFFLD information generated for the definition in the file object.

The *select-statement* must not reference variables, but may reference global variables.

The *select-statement* must not contain a PREVIOUS VALUE or a NEXT VALUE expression. The UPDATE, SKIP LOCKED DATA, and USE AND KEEP EXCLUSIVE LOCKS clauses may not be specified.

If the *select-statement* contains an *isolation-clause* the isolation level specified in the *isolation-clause* applies to the entire SQL statement.

WITH DATA
Specifies that the *select-statement* is executed. After the table is created, the result table rows of the *select-statement* are automatically inserted into the table.

WITH NO DATA
Specifies that the *select-statement* is used only to define the attributes of the new table. The table is not populated using the results of the *select-statement*.

unique-constraint

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that was previously specified in the CREATE TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

PRIMARY KEY(*column-name***,...)**
Defines a primary key composed of the identified columns. A table can only have one primary key. Thus, this clause cannot be specified more than once and cannot be specified at all if the shorthand form has been used to define a primary key for the table. The identified columns cannot be the same as the columns specified in another UNIQUE constraint specified earlier in the CREATE TABLE statement. For example, PRIMARY KEY(A,B) would not be allowed if UNIQUE (B,A) had already been specified.

Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB, DATALINK, or XML column. If a sort sequence is specified, the column must not contain a field procedure. The number of identified columns must not exceed 120, and the sum of their byte counts must

not exceed 32766-*n*, where *n* is the number of columns specified that allow nulls. For information about byte-counts see Table 77 on page 1028.

The unique index is created as part of the system physical file, not a separate system logical file. When a primary key is added, a CHECK constraint is implicitly added to enforce the rule that the NULL value is not allowed in any of the columns that make up the primary key.

UNIQUE (*column-name***,...)**
Defines a unique constraint composed of the identified columns. The UNIQUE clause can be specified more than once. The identified columns cannot be the same as the columns specified in another UNIQUE constraint or PRIMARY KEY that was specified earlier in the CREATE TABLE statement. For determining if a unique constraint is the same as another constraint specification, the column lists are compared. For example, UNIQUE (A,B) is the same as UNIQUE (B,A).

Each column-name must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB, DATALINK, or XML column. If a sort sequence is specified, the column must not contain a field procedure. The number of identified columns must not exceed 120, and the sum of their byte counts must not exceed 32766-*n*, where *n* is the number of columns specified that allows nulls. For information about byte-counts see Table 77 on page 1028.

A unique index on the identified column is created during the execution of the CREATE TABLE statement. The unique index is created as part of the system physical file, not as a separate system logical file.

referential-constraint

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that was previously specified in the CREATE TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

FOREIGN KEY
Each specification of the FOREIGN KEY clause defines a referential constraint. FOREIGN KEY is not allowed if the table is a declared global temporary table or a distributed table.

(*column-name***,...)**
The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB, DATALINK, or XML column and must not be a row change timestamp column. If a sort sequence is specified, the column must not contain a field procedure. The number of identified columns must not exceed 120, and the sum of their lengths must not exceed 32766-*n*, where *n* is the number of columns specified that allow nulls.

REFERENCES *table-name*
The *table-name* specified in a REFERENCES clause must identify the table being created or a base table that already exists at the application server, but it must not identify a catalog table, a declared temporary table, or a

distributed table. If the parent is a partitioned table, the unique index that enforces the parent unique constraint must be non-partitioned.

A referential constraint is a *duplicate* if its foreign key, parent key, and parent table are the same as the foreign key, parent key, and parent table of a previously specified referential constraint. Duplicate referential constraints are allowed, but not recommended.

Let T2 denote the identified parent table and let T1 denote the table being created.

The specified foreign key must have the same number of columns as the parent key of T2. The description of the *n*th column of the foreign key and the description of the *n*th column of that parent key must have identical data types, lengths, CCSIDs, and FIELDPROCs.

(*column-name***,...)**
The parent key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The same column must not be identified more than once. The column must not be a LOB, DATALINK, or XML column and must not be a row change timestamp column. If a sort sequence is specified, the column must not contain a field procedure. The number of identified columns must not exceed 120, and the sum of their byte counts must not exceed 32766-*n*, where *n* is the number of columns specified that allow nulls. For information about byte-counts see Table 77 on page 1028.

The list of column names must be identical to the list of column names in the primary key of T2 or a UNIQUE constraint that exists on T2. The names need not be specified in the same order as in the primary key; however, they must be specified in corresponding order to the list of columns in the *foreign key* clause. If a column name list is not specified, then T2 must have a primary key. Omission of the column name list is an implicit specification of the columns of that primary key.

The referential constraint specified by a FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent.

ON DELETE
Specifies what action is to take place on the dependent tables when a row of the parent table is deleted. There are five possible actions:

- NO ACTION (default)
- RESTRICT
- CASCADE
- SET NULL
- SET DEFAULT

SET NULL must not be specified unless some column of the foreign key allows null values.

CASCADE must not be specified if T1 contains a DataLink column with FILE LINK CONTROL.

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null. SET NULL is not allowed if the dependent table is a partitioned table and a foreign key column is also a partitioning key.
- If SET DEFAULT is specified, each column of the foreign key of each dependent of *p* in T1 is set to its default value. SET DEFAULT is not allowed if the dependent table is a partitioned table and a foreign key column is also a partitioning key.

ON UPDATE
Specifies what action is to take place on the dependent tables when a row of the parent table is updated.

The update rule applies when a row of T2 is the object of an UPDATE or propagated update operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are updated.

check-constraint

CONSTRAINT *constraint-name*
Names the check constraint. A *constraint-name* must not identify a constraint that was previously specified in the CREATE TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

CHECK (*check-condition***)**
Defines a check constraint. At any time, the *check-condition* must be true or unknown for every row of the table.

The *check-condition* is a *search-condition* except:

- It can only refer to columns of the table
- The result of any expression in the *check-condition* cannot be a ROWID, XML, or DATALINK with FILE LINK CONTROL data type. It cannot reference any column that has a field procedure, unless the column is referenced in a NULL predicate.
- It must not contain any of the following:
  - Subqueries
  - Aggregate functions
  - Variables
  - Global variables
  - Parameter markers
  - *Sequence-references*
  - Complex expressions that contain LOBs (such as concatenation)
  - OLAP specifications
  - ROW CHANGE expressions
  - REGEXP_LIKE predicate

- Special registers
- User-defined functions other than functions that were implicitly generated with the creation of a distinct type
- The following built-in scalar functions:

ATAN2	DLURLCOMPLETE [1]	LPAD	REPLACE
CARDINALITY	DLURLPATH	MAX_CARDINALITY	ROUND_TIMESTAMP
CONTAINS	DLURLPATHONLY	MONTHNAME	RPAD
CURDATE	DLURLSCHEME	MONTHS_BETWEEN	SCORE
CURTIME	DLURLSERVER	NEXT_DAY	SOUNDEX
DATAPARTITIONNAME	DLVALUE	NOW	TIMESTAMP_FORMAT
DATAPARTITIONNUM	ENCRYPT_AES	OVERLAY	TIMESTAMPDIFF
DAYNAME	ENCRYPT_RC2	RAISE_ERROR	TRUNC_TIMESTAMP
DBPARTITIONNAME	ENCRYPT_TDES	RAND	VARCHAR_FORMAT
DECRYPT_BINARY	GENERATE_UNIQUE	REGEXP_COUNT	WEEK_ISO
DECRYPT_BIT	GETHINT	REGEXP_INSTR	XMLPARSE
DECRYPT_CHAR	IDENTITY_VAL_LOCAL	REGEXP_REPLACE	XMLVALIDATE
DECRYPT_DB	INSERT	REGEXP_SUBSTR	XSLTRANSFORM
DIFFERENCE	LOCATE_IN_STRING	REPEAT	

[1] For DataLinks with an attribute of FILE LINK CONTROL and READ PERMISSION DB.

For more information about search-condition, see "Search conditions" on page 225. For more information about check constraints involving LOB data types and expressions, see the Database Programming topic collection.

NOT LOGGED INITIALLY

Any changes made to the table by INSERT, DELETE, or UPDATE statements in the same unit of work after the table is created by this statement are not logged (journaled).

At the completion of the current unit of work, the NOT LOGGED INITIALLY attribute is deactivated and all operations that are done on the table in subsequent units of work are logged (journaled).

The NOT LOGGED INITIALLY option is useful for situations where a large result set needs to be created with data from an alternate source (another table or a file) and recovery of the table is not necessary. Using this option will save the overhead of logging (journaling) the data.

ACTIVATE NOT LOGGED INITIALLY is ignored if the table has a DATALINK column with FILE LINK CONTROL.

VOLATILE or NOT VOLATILE

Indicates to the optimizer whether the cardinality of table *table-name* can vary significantly at run time. Volatility applies to the number of rows in the table, not to the table itself. The default is NOT VOLATILE.

VOLATILE
Specifies that the cardinality of *table-name* can vary significantly at run time, from empty to large. To access the table, the optimizer will typically use an index, if possible.

NOT VOLATILE
Specifies that the cardinality of *table-name* is not volatile. Access plans that

reference this table will be based on the cardinality of the table at the time the access plan is built. NOT VOLATILE is the default.

RCDFMT

Indicates the record format name of the table.

RCDFMT *format-name*
An unqualified name that designates the IBM i record format name of the table. A *format-name* is a system identifier.

If a record format name is not specified, the *format-name* is the same as the *system-object-name* of the table.

media-preference

Specifies the preferred storage media for the table or partition.

UNIT ANY
No storage media is preferred. Storage for the table or partition will be allocated from any available storage media. If UNIT ANY is specified on the table, any *media-preference* that is specified on a partition is used.

UNIT SSD
Solid state disk storage media is preferred. Storage for the table or partition may be allocated from solid state disk storage media, if available. If UNIT SSD is specified on the table, any *media-preference* specified on a partition is ignored.

ON REPLACE

Specifies the action to take when replacing a table that exists at the current server. This option is ignored if an existing table is not being replaced.

PRESERVE ALL ROWS
Current rows of the specified table will be preserved. PRESERVE ALL ROWS is not allowed if WITH DATA is specified with *result-table-as*.

All rows of all partitions in a partitioned table are preserved. If the new table definition is a range partitioned table, the defined ranges must be able to contain all the rows from the existing partitions.

If a column is dropped, the column values will not be preserved. If a column is altered, the column values may be modified.

If the table is not a partitioned table or is a hash partitioned table, PRESERVE ALL ROWS and PRESERVE ROWS are equivalent.

PRESERVE ROWS
Current rows of the specified table will be preserved. PRESERVE ROWS is not allowed if WITH DATA is specified with *result-table-as*.

If a partition of a range partitioned table is dropped, the rows of that partition will be deleted without processing any delete triggers. To determine if a partition of a range partitioned table is dropped, the range definitions and the partition names (if any) of the partitions in the new table definition are compared to the partitions in the existing table definition. If either the specified range or the name of a partition matches, it is preserved. If a partition does not have a *partition-name*, its *boundary-spec* must match an existing partition.

If a partition of a hash partition table is dropped, the rows of that partition will be preserved.

If a column is dropped, the column values will not be preserved. If a column is altered, the column values may be modified.

DELETE ROWS
Current rows of the specified table will be deleted. Any existing DELETE triggers are not fired.

distribution-clause

IN NODEGROUP *nodegroup-name*
Specifies the nodegroup across which the data in the table will be distributed. The name must identify a nodegroup that exists at the current server. If this clause is specified, the table is created as a distributed table across all the systems in the nodegroup.

A LOB, DATALINK, XML, or IDENTITY column is not allowed in a distributed table.

The DB2 Multisystem product must be installed to create a distributed table. For more information about distributed tables, see the DB2 Multisystem topic collection.

DISTRIBUTE BY HASH (*column-name***,...)**
Specifies the partitioning key. The partitioning key is used to determine on which node in the nodegroup a row will be placed. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. If the DISTRIBUTE BY clause is not specified, the first column of the primary key is used as the partitioning key. If there is no primary key, the first column of the table that is not floating point, date, time, or timestamp is used as the partitioning key.

The columns that make up the partitioning key must be a subset of the columns that make up any unique constraints over the table. Floating point, date, time, timestamp, LOB, XML, DataLink, ROWID, and columns that have a field procedure cannot be used in a partitioning key.

partitioning-clause

PARTITION BY RANGE
Specifies that ranges of column values are used to determine the target data partition when inserting a row into the table. The number of partitions must not exceed 256.

partition-expression
Specifies the key data over which the range is defined to determine the target data partition of the data.

column-name
Identifies a column in the data partitioning key. The partitioning key is used to determine into which partition in the table a row will be placed. The *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once.

LOB, XML, DataLink, ROWID, row change timestamp columns, identity columns, and any column with a field procedure cannot be used in a partitioning key.

The number of identified columns must not exceed 120. The sum of length attributes of the columns must not be greater than 2000.

NULLS LAST
Indicates that null values compare high.

NULLS FIRST
Indicates that null values compare low.

partition-element
Specifies ranges for a data partitioning key.

PARTITION *partition-name*
Names the data partition. The name must not be the same as any other data partition for the table.

If the clause is not specified, a unique partition name is generated by the database manager.

boundary-spec
Specifies the boundaries of a range partition. If more than one partition key is specified, the boundaries must be specified in ascending sequence. The ranges must not overlap.

starting-clause
Specifies the low end of the range for a data partition. The number of specified starting values must be the same as the number of columns in the partitioning key. If a *starting-clause* is not specified for the first *boundary-spec*, the default is MINVALUE INCLUSIVE for each column of the partitioning key. If a *starting-clause* is not specified for a subsequent *boundary-spec*, the previous adjacent *boundary-spec* must contain an *ending-clause*. The default is the same as that *ending-clause* except that the INCLUSIVE or EXCLUSIVE attribute is reversed.

STARTING FROM
Introduces the *starting-clause*.

constant
Specifies a constant that must conform to the rules of a constant for the data type of the corresponding column of the partition key. If the corresponding column of the partition key is a distinct type, the constant must conform to the rules of the source type of the distinct type. The value must not be in the range of any other *boundary-spec* for the table.

MINVALUE
Specifies a value that is lower than the lowest possible value for the data type of the *column-name* to which it corresponds. If MINVALUE is specified, all subsequent values in the *starting-clause* must also be MINVALUE.

MAXVALUE
Specifies a value that is greater than the greatest possible value for the data type of the *column-name* to which it corresponds. If MAXVALUE is specified, all subsequent values in the *ending-clause* must also be MAXVALUE.

INCLUSIVE
Indicates that the specified range values are included in the data partition.

EXCLUSIVE
Indicates that the specified range values are excluded from the data partition. This specification is ignored when MINVALUE or MAXVALUE is specified.

ending-clause
Specifies the high end of the range for a data partition. The number of specified ending values must be the same as the number of columns in the data partitioning key. An *ending-clause* must be specified for the last *boundary-spec*. If an *ending-clause* is not specified for a previous *boundary-spec*, the next adjacent *boundary-spec* must contain a *starting-clause*. The default is the same as that *starting-clause* except that the INCLUSIVE or EXCLUSIVE attribute is reversed.

ENDING AT
Introduces the *ending-clause*.

constant
Specifies a constant that must conform to the rules of a constant for the data type of the corresponding column of the partition key. If the corresponding column of the partition key is a distinct type, the constant must conform to the rules of the source type of the distinct type. The value must not be in the range of any other *boundary-spec* for the table.

MINVALUE
Specifies a value that is lower than the lowest possible value for the data type of the *column-name* to which it corresponds. If MINVALUE is specified, all subsequent values in the *starting-clause* must also be MINVALUE.

MAXVALUE
Specifies a value that is greater than the greatest possible value for the data type of the *column-name* to which it corresponds. If MAXVALUE is specified, all subsequent values in the *ending-clause* must also be MAXVALUE.

INCLUSIVE
Indicates that the specified range values are included in the data partition.

EXCLUSIVE
Indicates that the specified range values are excluded from the data partition. This specification is ignored when MINVALUE or MAXVALUE is specified.

EVERY *integer-constant*
Specifies that multiple data partitions will be added where *integer-constant* specifies the width of each data partition range. If EVERY is specified, only a single SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, DATE, or TIMESTAMP column can be specified for the partition key.

The starting value of the first data partition is the specified STARTING value. The starting value of each subsequent partition is the starting value of the previous partition + *integer-constant*. If the *starting-clause*

specified EXCLUSIVE, the starting value of every partition is EXCLUSIVE. Otherwise, the starting value of every partition is INCLUSIVE.

The ending value of every partition of the range is (start + *integer-constant* - 1). If the *ending-clause* specified EXCLUSIVE, the ending value of every partition is EXCLUSIVE. Otherwise, the ending value of every partition is INCLUSIVE.

The number of partitions added is determined by adding *integer-constant* repeatedly to the STARTING value until the ENDING value is reached. For example:

```
CREATE TABLE FOO
  (A  INT)
  PARTITION BY RANGE(A)
    (STARTING(1) ENDING(10) EVERY(2))
```

is equivalent to the following CREATE TABLE statement:

```
CREATE TABLE FOO
  (A  INT)
  PARTITION BY RANGE(A)
    (STARTING(1) ENDING(2),
     STARTING(3) ENDING(4),
     STARTING(5) ENDING(6),
     STARTING(7) ENDING(8),
     STARTING(9) ENDING(10))
```

In the case of dates and timestamps, the EVERY value must be a labeled duration. For example:

```
CREATE TABLE FOO
  (A  DATE)
  PARTITION BY RANGE(A)
    (STARTING('2001-01-01') ENDING('2010-01-01') EVERY(3 MONTHS))
```

PARTITION BY HASH
Specifies that the hash function is used to determine the target data partition when inserting a row into the table.

(*column-name***,...)**
Specifies the partitioning key. The partitioning key is used to determine into which partition in the table a row will be placed. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once.

The columns that make up the partitioning key must be a subset of the columns that make up any unique constraints over the table. Floating point, decimal floating-point, LOB, XML, date, time, timestamp, DataLink, ROWID, row change timestamp columns, identity columns, and columns with a field procedure cannot be used in a partitioning key.

INTO *integer* **PARTITIONS**
Specifies the number of partitions. The number of partitions must not exceed 256.

materialized-query-definition

column-name
Names a column in the table. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the *select-statement*. Each *column-name* must be unique and unqualified. If a list of

column names is not specified, the columns of the table inherit the names of the columns of the result table of the *select-statement*.

A list of column names must be specified if the result table of the *select-statement* has duplicate column names or an unnamed column. An unnamed column is a column derived from a constant, function, expression, or set operation (UNION or INTERSECT) that is not named using the AS clause of the select list.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the *system-column-name* is not specified, and the column-name is not a valid *system-column-name*, a system column name is generated. For more information about how system column names are generated, see "Rules for Table Name Generation" on page 1030.

select-statement
Specifies that the columns of the table have the same name and description as the columns that would appear in the derived result table of the *select-statement* if the *select-statement* were to be executed. The use of AS *(select-statement)* is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *select-statement*.

The implicit definition includes the following attributes of the n columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID
- Nullability
- FIELDPROC
- Column heading and text (see "LABEL" on page 1263)

The following attributes are not included:

- Default value
- Hidden attribute
- Identity attributes
- Row change timestamp attribute

The implicit definition does not include any other optional attributes of the identified table or view. For example, the new table does not automatically include a primary key or foreign key from a table. The new table has these and other optional attributes only if the optional clauses are explicitly specified.

The *select-statement* must not refer to variables or global variables, or include parameter markers. If an expression in the SELECT clause of the *select-statement* is not a column name, then the expression must not reference a column with a field procedure.

The *select-statement* must not contain a PREVIOUS VALUE or a NEXT VALUE expression. The UPDATE, SKIP LOCKED DATA, and USE AND KEEP EXCLUSIVE LOCKS clauses may not be specified.

refreshable-table-options
Specifies that the table is a *materialized query table* and the REFRESH TABLE statement can be used to populate the table with the results of the *select-statement*.

A materialized query table whose *select-statement* contains a GROUP BY clause is summarizing data from the tables referenced in the *select-statement*. Such a materialized query table is also known as a *summary table*. A summary table is a specialized type of materialized query table.

When a materialized query table is defined, the following *select-statement* restrictions apply:

- The *select-statement* cannot contain a reference to another materialized query table or to a view that refers to a materialized query table.
- The *select-statement* cannot contain a reference to a declared temporary table, a table in QTEMP, a program-described file, or a non-SQL logical file in the FROM clause.
- The *select-statement* cannot contain a *data-change-file-reference*.
- The *select-statement* cannot contain a reference to a view that references another materialized query table or a declared temporary table. When a materialized query table is defined with ENABLE QUERY OPTIMIZATION, the *select-statement* cannot contain a reference to a view that contains one of the restrictions from the following paragraph.
- The *select-statement* cannot contain an expression with a DataLink or a distinct type based on a DataLink where the DataLink is FILE LINK CONTROL.
- The *select-statement* cannot contain a result column that is a not an SQL data type, such as binary with precision, DBCS-ONLY, or DBCS-EITHER.

When a materialized query table is defined with ENABLE QUERY OPTIMIZATION, the following additional *select-statement* restrictions apply:

- Must not include any special registers.
- Must not include any non-deterministic functions.
- The ORDER BY clause is allowed, but is only used by REFRESH. It may improve locality of reference of data in the materialized query table.
- If the subselect references a view, the *select-statement* in the view definition must satisfy the preceding restrictions.

DATA INITIALLY DEFERRED
Specifies that the data is not inserted into the materialized query table when it is created. Use the REFRESH TABLE statement to populate the materialized query table, or use the INSERT statement to insert data into a materialized query table.

DATA INITIALLY IMMEDIATE
Specifies that the data is inserted into the materialized query table when it is created.

REFRESH DEFERRED
Specifies that the data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time when the REFRESH TABLE statement is processed or when it was last updated.

MAINTAINED BY USER
Specifies that the materialized query table is maintained by the user. The user can use INSERT, DELETE, UPDATE, or REFRESH TABLE statements on the table.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether this materialized query table can be used for optimization. The default is ENABLE QUERY OPTIMIZATION.

ENABLE QUERY OPTIMIZATION
Specifies that the materialized query table can be used for query optimization. If the *select-statement* specified does not satisfy the restrictions for query optimization, an error is returned.

DISABLE QUERY OPTIMIZATION
Specifies that the materialized query table cannot be used for query optimization. The table can still be queried directly.

Notes

Table attributes: Tables are created as physical files. When a table is created, the file wait time and record wait time attributes are set to the default that is specified on the WAITFILE and WAITRCD keywords of the Create Physical File (CRTPF) command.

SQL tables are created so that space used by deleted rows will be reclaimed by future insert requests. This attribute can be changed via the command *CHGPF* and specifying the *REUSEDLT(*NO)* parameter. For more information about the *CHGPF* command, see CL Reference.

A distributed table is created on all of the servers across which the table is distributed. For more information about distributed tables, seeDB2 Multisystem.

Table journaling: When a table is created, journaling may be automatically started.

- If a data area called QDFTJRN exists in the same schema that the table is created into and the user is authorized to the data area, journaling will be started to the journal named in the data area if all the following are true:
  - The identified schema for the table must not be QSYS, QSYS2, QRECOVERY, QSPL, QRCL, QRPLOBJ, QGPL, QTEMP, SYSIBM, or any of the iASP equivalents to these libraries.
  - The journal specified in the data area must exist and the user must be authorized to start journaling to the journal.
  - The first 10 bytes of the data area must contain the name of the schema in which to find the journal.
  - The second 10 bytes must contain the name of the journal.
  - The remaining bytes contain the object types being implicitly journaled and the options that affect when implicit journaling is performed. The object type must include the value *FILE or *ALL. The value *NONE can be used to prevent journaling from being started.

  For more information, see Journal Management.
- If the table is created into a schema that has specified (using the STRJRNLIB command) that journaling should implicitly be started.
- If a data area called QDFTJRN does not exist in the same schema that the table is created into or the user is not authorized to the data area and the schema has not specified that journaling should be started, journaling will be started to a journal called QSQJRN if it exists in the same schema that the table is created into.

Table ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the table is created exists, the *owner* of the table is that user profile.

- Otherwise, the *owner* of the table is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the table is the user profile or group user profile of the job executing the statement.

Table authority: If SQL names are used, tables are created with the system authority of *EXCLUDE to *PUBLIC. If system names are used, tables are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the table is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the table.

REPLACE rules: When a table is recreated by REPLACE using PRESERVE ROWS, the new definition of the table is compared to the old definition and logically, for each difference between the two, a corresponding ALTER operation is performed. When the DELETE ROWS option is used, the table is logically dropped and recreated; as long as objects dependent on the table remain valid, any modification is allowed. For more information see Table 72 on page 790 in ALTER TABLE. For columns, constraints, and partitions, the comparisons are performed based on their names and attributes.

Column names and system column names can be changed by modifying either the column name or the system column name, leaving the other name unchanged. If neither name matches an existing column, a new column is created. The existence of another object dependent on a column name may prevent the name change.

New definition vs existing definition	**Equivalent ALTER TABLE action**
Column	
Column exists in both and the attributes are the same	No change
Column exists in both and the attributes are different	ALTER COLUMN
Column exists only in new table definition	ADD COLUMN
Column exists only in existing table definition	DROP COLUMN RESTRICT
Constraint	
Constraint exists in both and is the same	No change
Constraint exists in both and is different	DROP *constraint* RESTRICT and ADD *constraint*
Constraint exists only in new table definition	ADD *constraint*
Constraint exists only in existing table definition	DROP *constraint* RESTRICT
materialized-query-definition	
materialized-query-definition exists in both and is the same	No change
materialized-query-definition exists in both and is different	ALTER MATERIALIZED QUERY
materialized-query-definition exists only in new table definition	ADD MATERIALIZED QUERY

New definition vs existing definition	Equivalent ALTER TABLE action
materialized-query-definition exists only is existing table definition	DROP MATERIALIZED QUERY
partitioning-clause	
partitioning-clause exists in both and is the same	No change
partitioning-clause exists in both and is different	ADD PARTITION, DROP PARTITION, and ALTER PARTITION
partitioning-clause exists only in new table definition	ADD *partitioning-clause*
partitioning-clause exists only in existing table definition	DROP PARTITIONING
NOT LOGGED INITIALLY	
NOT LOGGED INITIALLY exists in both	No change
NOT LOGGED INITIALLY exists only in new table definition	NOT LOGGED INITIALLY
NOT LOGGED INITIALLY exists only in existing table definition	Logged initially
VOLATILE	
VOLATILE attribute exists in both and is the same	No change
VOLATILE attribute exists only in new table definition	VOLATILE
VOLATILE attribute exists only in existing table definition	NOT VOLATILE
media-preference	
media-preference exists in both and is the same	No change
media-preference exists only in new table definition	ALTER *media-preference*
media-preference exists only in existing table definition	UNIT ANY

Any attributes that cannot be specified in the CREATE TABLE statement are preserved:

- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved. However, unlike other objects, REPLACE of a table will generate a ZC (change object) journal audit entry.
- Current data journaling is preserved.
- Comments and labels are preserved.
- Triggers are preserved, if possible. If it is not possible to preserve a trigger, an error is returned.
- Any views, materialized query tables, and indexes dependent on the table will be preserved or recreated, if possible. If it is not possible to preserve a dependent view, materialized query table, or index, an error is returned.

Using an identity column: When a table has an identity column, the database manager can automatically generate sequential numeric values for the column as rows are inserted into the table. Thus, identity columns are ideal for primary keys.

Identity columns and ROWID columns are similar in that both types of columns contain values that the database manager generates. ROWID columns can be useful in direct-row access. ROWID columns contain values of the ROWID data type, which returns a 40-byte VARCHAR value that is not regularly ascending or descending. ROWID data values are therefore not well suited to many application uses, such as generating employee numbers or product numbers. For data that does not require direct-row access, identity columns are usually a better approach, because identity columns contain existing numeric data types and can be used in a wide variety of uses for which ROWID values would not be suitable.

When a table is recovered to a point-in-time (using RMVJRNCHG), it is possible that a large gap in the sequence of generated values for the identity column might result. For example, assume a table has an identity column that has an incremental value of 1 and that the last generated value at time T1 was 100 and the database manager subsequently generates values up to 1000. Now, assume that the table is recovered back to time T1. The generated value of the identity column for the next row that is inserted after the recovery completes will be 1001, leaving a gap from 100 to 1001 in the values of the identity column.

When CYCLE is specified duplicate values for a column may be generated even when the column is GENERATED ALWAYS, unless a unique constraint or unique index is defined on the column.

Creating materialized query tables: To ensure that the materialized query table has data before being used by a query:

- DATA INITIALLY IMMEDIATE should be used to create materialized query tables, or
- the materialized query table should be created with query optimization disabled and then enable the table for query optimization after it is refreshed.

The isolation level at the time when the CREATE TABLE statement is executed is the isolation level for the materialized query table. The *isolation-clause* can be used to explicitly specify the isolation level.

Considerations for implicitly hidden columns: A column that is defined as implicitly hidden is not part of the result table of a query that specifies * in a SELECT list. However, an implicitly hidden column can be explicitly referenced in a query. For example, an implicitly hidden column can be referenced in the SELECT list or in a predicate in a query. Additionally, an implicitly hidden column can be explicitly referenced in a COMMENT statement, CREATE INDEX statement, ALTER TABLE statement, INSERT statement, MERGE statement, or UPDATE statement. An implicitly hidden column can be referenced in a referential constraint. A REFERENCES clause that does not contain a column list refers implicitly to the primary key of the parent table. It is possible that the primary key of the parent table includes a column defined as implicitly hidden. Such a referential constraint is allowed.

If the SELECT list of the fullselect of a materialized query definition explicitly refers to an implicitly hidden column, that column will be part of the materialized query table.

If the SELECT list of the fullselect of a view definition (CREATE VIEW statement) explicitly refers to an implicitly hidden column, that column will be part of the view, however the view column is not considered 'hidden'.

Partitioned table performance: The larger the number of partitions in a partitioned table, the greater the overhead in SQL data change and SQL data statements. You should create a partitioned table with the minimum number of partitions that are required to minimize this overhead. It is also highly recommended that a parallelism degree greater than one be considered when accessing a partitioned table.

Creating a table using a remote select-statement: The *select-statement* for an *as-result-table* can refer to tables on a different server than where the table is being created. This can be done by using a three-part object name or an alias that is defined to reference a three-part name of a table or view. The *select-statement* cannot be for a materialized query table and the result cannot contain a column that has a field procedure defined.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- INLINE LENGTH is a synonym for ALLOCATE.
- *constraint-name* (without the CONSTRAINT keyword) may be specified following the FOREIGN KEY keywords in a *referential-constraint*
- DEFINITION ONLY is a synonym for WITH NO DATA
- PARTITIONING KEY is a synonym for DISTRIBUTE BY HASH.
- PART is a synonym for PARTITION.
- PARTITION *partition-number* may be specified instead of PARTITION *partition-name*. A *partition-number* must not identify a partition that was previously specified in the CREATE TABLE statement.

  If a *partition-number* is not specified, a unique partition number is generated by the database manager.
- VALUES is a synonym for ENDING AT.
- SUMMARY between CREATE and TABLE when creating a materialized query table.

Maximum row sizes

There are two maximum row size restrictions referred to in the description of *column-definition*.

- The maximum row buffer size is 32766 or, if a VARCHAR, VARGRAPHIC, VARBINARY, LOB, or XML column is specified, 32740.
- The maximum row data size is 3 758 096 383 if a LOB or XML column is specified; this size is determined when a row is inserted or updated. If a LOB or XML column is not specified, then the maximum row data size is 32766 or, if a VARCHAR, VARGRAPHIC, or VARBINARY column is specified, 32740.

To determine the length of a row buffer and row data or both, add the corresponding length of each column of that row based on the byte counts of the data type.

The follow table gives the byte counts of columns by data type for columns that do not allow null values. If any column allows null values, one byte is required for

every eight columns. A column that has a field procedure could have a different count based on the result of the field procedure.

Table 77. Byte Counts of Columns by Data Type

Data Type	Row Buffer Byte Count	Row Data Byte Count
SMALLINT	2	2
INTEGER	4	4
BIGINT	8	8
DECIMAL(*p*, *s*)	The integral part of (*p*/2) + 1	The integral part of (*p*/2) + 1
NUMERIC(*p*, *s*)	*p*	*p*
FLOAT (single precision)	4	4
FLOAT (double precision)	8	8
DECFLOAT(16)	8	8
DECFLOAT(34)	16	16
CHAR(*n*)	*n*	*n*
VARCHAR(*n*)	*n*+2	*n*+2
CLOB(*n*)	29+*pad*	*n*+29
GRAPHIC(*n*)	*n**2	*n**2
VARGRAPHIC (n)	*n**2+2	*n**2+2
DBCLOB(*n*)	29+*pad*	*n**2+29
BINARY(*n*)	*n*	*n*
VARBINARY(*n*)	*n*+2	*n*+2
BLOB(*n*)	29+*pad*	*n*+29
DATE	10	4
TIME	8	3
TIMESTAMP	26	10
DATALINK(*n*)	*n*+24	*n*+24
ROWID	42	28
XML	29+*pad*	2 147 483 647
distinct-type	The byte count for the source type.	The byte count for the source type.
Notes: *pad* is a value from 1 to 15 necessary for boundary alignment.		

Precision as described to the database:

- Floating-point fields are defined in the DB2 for i database with a decimal precision, not a bit precision. The algorithm used to convert the number of bits to decimal is *decimal precision = CEILING(n/3.31)*, where *n* is the number of bits to convert. The decimal precision is used to determine how many digits to display using interactive SQL.
- SMALLINT fields are stored with a decimal precision of 4,0.
- INTEGER fields are stored with a decimal precision of 9,0.
- BIGINT fields are stored with a decimal precision of 19,0.

LONG VARCHAR, LONG VARGRAPHIC, and LONG VARBINARY

The non-standard syntax of LONG VARCHAR, LONG VARGRAPHIC, and LONG VARBINARY is supported, but deprecated. The alternative standard syntax of VARCHAR(integer), VARGRAPHIC(integer), and VARBINARY(integer) is preferred. VARCHAR(integer), VARGRAPHIC(integer), and VARBINARY(integer) are recommended. After the CREATE TABLE statement is processed, the database manager considers a LONG VARCHAR column to be VARCHAR, a LONG VARGRAPHIC column to be VARGRAPHIC, and a LONG VARBINARY column to be VARBINARY. The maximum length is calculated in a product-specific fashion that is not portable.

LONG VARCHAR [95]
For a varying length character string whose maximum length is determined by the amount of space available in the row.

LONG VARGRAPHIC [95]
For a varying length graphic string whose maximum length is determined by the amount of space available in the row.

LONG VARBINARY [95]
For a varying length binary string whose maximum length is determined by the amount of space available in the row.

The maximum length of a LONG column is determined as follows. Let:

- `i` be the sum of the row buffer byte counts of all columns in the table that are not LONG VARCHAR, LONG VARGRAPHIC, or LONG VARBINARY
- `j` be the number of LONG VARCHAR, LONG VARGRAPHIC, and LONG VARBINARY columns in the table
- `k` be the number of columns in the row that allow nulls.

The length of each LONG VARCHAR and LONG VARBINARY column is INTEGER((32716 - `i`-((`k`+7)/8))/`j`).

The length of each LONG VARGRAPHIC column is determined by taking the length calculated for a LONG VARCHAR column and dividing it by 2. The integer portion of the result is the length.

Rules for System Name Generation

There are specific instances when the system generates a system table, view, index, or column name. These instances and the name generation rules are described in the following sections.

Rules for Column Name Generation

A system-column-name is generated if the system-column-name is not specified when a table or view is created and the column-name is not a valid system-column-name.

If the column-name does not contain special characters and is longer than 10 characters, a 10-character system-column-name will be generated as:

- The first 5 characters of the name

95. This option is provided for compatibility with other products. It is recommended that VARCHAR(integer), VARGRAPHIC(integer), or VARBINARY(integer) be specified instead.

- A 5 digit unique number

For example:

```
The system-column-name for LONGCOLUMNNAME would be LONGC00001
```

If the column name is delimited:

- The first 5 characters from within the delimiters will be used as the first 5 characters of the system-column-name. If there are fewer than 5 characters within the delimiters, the name will be padded on the right with underscore (_) characters. Lower case characters are folded to upper case characters. The only valid characters in a system-column-name are: A-Z, 0-9, @, #, $, and _. Any other characters will be changed to the underscore (_) character. If the first character ends up as an underscore, it will be changed to the letter Q.
- A 5 digit unique number is appended to the 5 characters.

For example:

```
The system-column-name for "abc" would be ABC__00001
The system-column-name for "COL2.NAME" would be COL2_00001
The system-column-name for "C 3" would be C_3__00001
The system-column-name for "??" would be Q____00001
The system-column-name for "*column1" would be QCOLU00001
```

Rules for Table Name Generation

A system name will be generated if a table, view, alias, or index is created without using the FOR SYSTEM NAME clause and has either:

- A name longer than 10 characters
- A name that contains characters not valid in a system name

The SQL name or its corresponding system name may both be used in SQL statements to access the file once it is created. However, the SQL name is only recognized by DB2 for i and the system name must be used in other environments.

There are two separate methods for generating the system name:

- If a data area with the name QGENOBJNAM exists in the same schema that the table is created into, the user can influence the generated name.

  The data area is subject to the following restrictions:

  – The user must be authorized to read the data area.
  – The data area must have an attribute of CHAR(10).
  – The first 5 characters of the data area value must be '?????'.
  – The next 5 characters of the data area value must contain 5 numeric digits.

  If any of the above conditions are not satisfied or any error occurs while accessing the starting value in the data area, the default name generation rules will be used as if the data area did not exist at all.

  If the data area meets all of the restrictions above, the generated name will be the same as if the default name generation rules below except that after the first 5 (or 4) characters of the name, the unique number will initially contain the 5 digits specified in the data area (instead of '00001' or '0001').

  For example, if the value of the data area was '?????00999':

  ```
  The system name for "??" would be "__00999"
  The system name for "longtablename" would be "lon00999"
  The system name for "LONGTableName" would be LONG00999
  The system name for "A b   " would be "A_b00999"
  ```

- Otherwise, the default name generation rules are used:

  If the name does not contain special characters and is longer than 10 characters, a 10-character system name will be generated as:

  – The first 5 characters of the name
  – A 5 digit unique number

  For example:

  ```
  The system name for LONGTABLENAME would be LONGT00001
  ```

  If the SQL name contains special characters, the system name is generated as:

  – The first 4 characters of the name
  – A 4 digit unique number

  In addition:

  – All special characters are replaced by the underscore (_)
  – Any trailing blanks are removed from the name
  – The name is delimited by double quotes (") if the delimiters are required for the name to be a valid system name.

  For example:

  ```
  The system name for "??" would be "__0001"
  The system name for "longtablename" would be "long0001"
  The system name for "LONGTableName" would be LONG0001
  The system name for "A b   " would be "A_b0001"
  ```

  SQL ensures the system name is unique by searching the cross reference file. If the name already exists in the cross reference file, the number is incremented until the name is no longer a duplicate.

  If a unique name cannot be determined using the above rules, an additional character is added to the counter in the name, and the number is incremented until a unique name can be found or the range is exhausted. For example, if creating "longtablename" and names "long0001" through "long9999" already exist, the name would become "lon00001".

Examples

Example 1: Given administrative authority, create a table named 'ROSSITER.INVENTORY' with the following columns:

Part number
Small integer, must not be null

Description
Character of length 0 to 24, allows nulls

Quantity on hand,
Integer allows nulls

```
CREATE TABLE ROSSITER.INVENTORY
  (PARTNO         SMALLINT     NOT NULL,
   DESCR          VARCHAR(24),
   QONHAND        INT)
```

Example 2: Create a table named DEPARTMENT with the following columns:

Department number
Character of length 3, must not be null

Department name
Character of length 0 through 36, must not be null

Manager number
Character of length 6

Administrative dept.
Character of length 3, must not be null

Location name
Character of length 16, allows nulls

The primary key is column DEPTNO.

```
CREATE TABLE DEPARTMENT
  (DEPTNO    CHAR(3)     NOT NULL,
   DEPTNAME  VARCHAR(36) NOT NULL,
   MGRNO     CHAR(6),
   ADMRDEPT  CHAR(3)     NOT NULL,
   LOCATION  CHAR(16),
   PRIMARY KEY(DEPTNO))
```

Example 3: Create a table named REORG_PROJECTS which has the same column definitions as the columns in the view PRJ_LEADER.

```
CREATE TABLE REORG_PROJECTS
  LIKE PRJ_LEADER
```

Example 4: Create an EMPLOYEE2 table with an identity column named EMP_NO. Define the identity column so that DB2 for i will always generate the values for the column. Use the default value, which is 1, for the first value that should be assigned and for the incremental difference between the subsequently generated consecutive numbers.

```
CREATE TABLE EMPLOYEE2
  ( EMPNO INTEGER GENERATED ALWAYS AS IDENTITY,
    ID SMALLINT,
    NAME CHAR(30),
    SALARY DECIMAL(5,2),
    DEPTNO SMALLINT)
```

Example 5: Assume a very large transaction table named TRANS contains one row for each transaction processed by a company. The table is defined with many columns. Create a materialized query table for the TRANS table that contains daily summary data for the date and amount of a transaction.

```
CREATE TABLE STRANS
  AS (SELECT  YEAR AS SYEAR, MONTH AS SMONTH, DAY AS SDAY, SUM(AMOUNT) AS SSUM
      FROM TRANS
      GROUP BY YEAR, MONTH, DAY )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER
```

CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- Each of the following:
  - The ALTER privilege on the table or view on which the trigger is defined,
  - The SELECT privilege on the table or view on which the trigger is defined,
  - The SELECT privilege on any table or view referenced in the *search-condition* in the *trigger-action*,
  - The UPDATE privilege on the table on which the trigger is defined, if the BEFORE UPDATE trigger contains a SET statement that modifies the NEW correlation variable,
  - The privileges required to execute each *triggered-SQL-statement*, and
  - The system authority *EXECUTE on the library containing the table or view on which the trigger is defined.
- Administrative authority

If an INSTEAD OF trigger is added to a view that is not inherently updatable, the *OBJMGT system authority is also required on the view.

In addition, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - *USE on the Add Physical File Trigger (ADDPFTRG) command,
  - *USE on the Create Program (CRTPGM) command
- Administrative authority

If SQL names are specified, and a user profile exists that has the same name as the library into which the trigger is created, and the name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- *ALLOBJ and *SECADM special authority
- Administrative authority

To replace an existing trigger, the privileges held by the authorization ID of the statement must include at least one of the following:

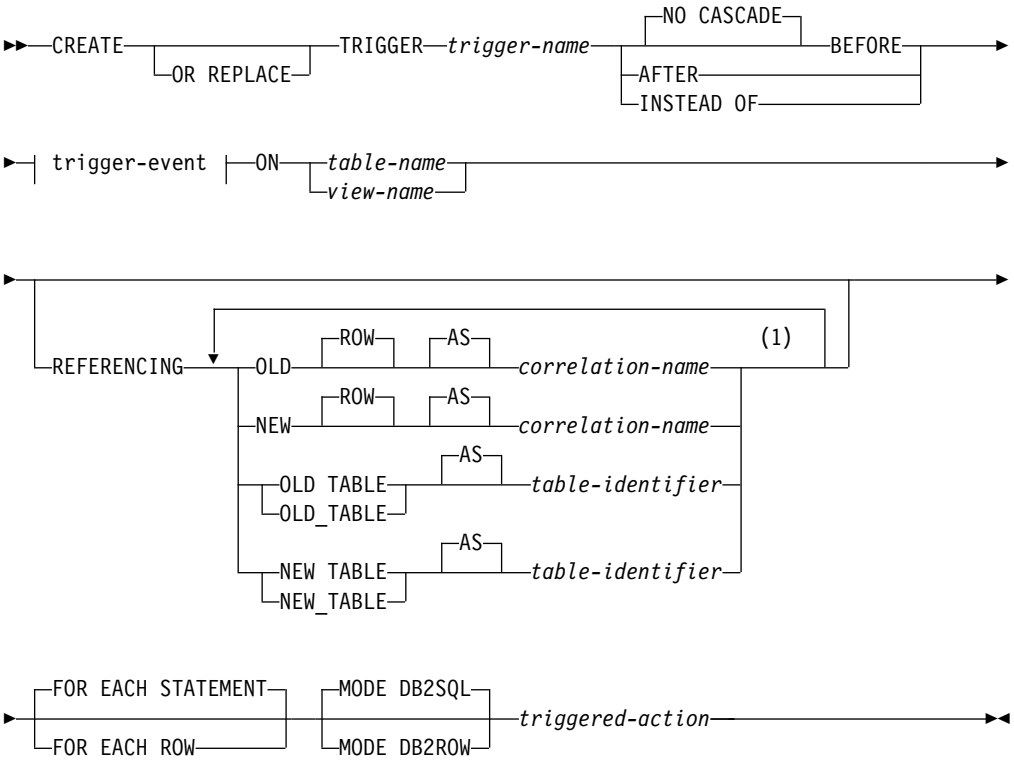
- The following system authorities:

- The system authority of *OBJMGT on the trigger program object
- All authorities needed to DROP the trigger

- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

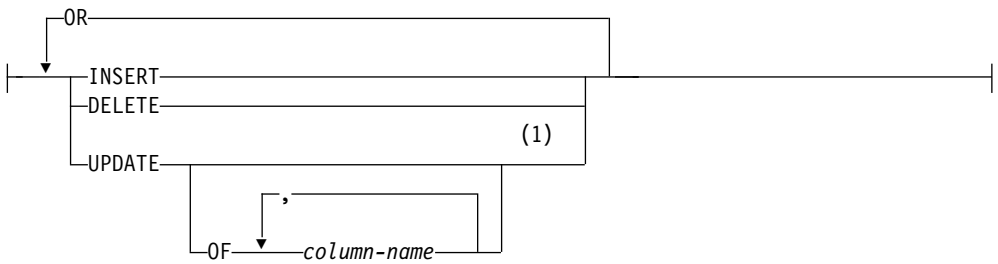
Syntax



Notes:

1 The same clause must not be specified more than once.

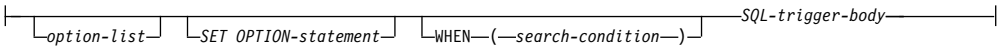
trigger-event:



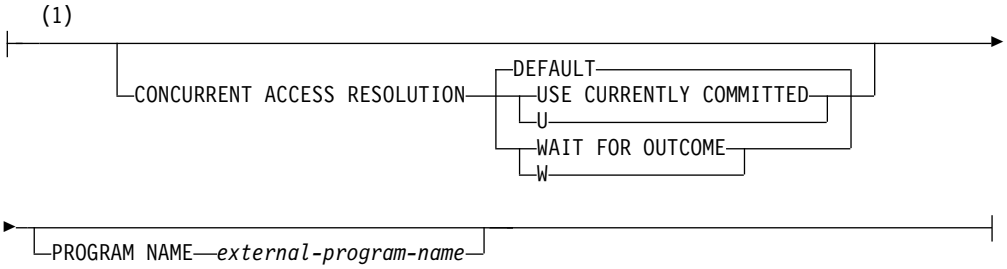
Notes:

1 Each *trigger-event* option can be specified only one time.

triggered-action:



option-list:



Notes:

1 The options in the option-list can be specified in any order.

SQL-trigger-body:

Description

OR REPLACE
Specifies to replace the definition for the trigger if one exists at the current

server. The existing definition is effectively dropped before the new definition is replaced in the catalog. This option is ignored if a definition for the trigger does not exist at the current server.

trigger-name

Names the trigger. The name, including the implicit or explicit qualifier, must not be the same as a trigger that already exists at the current server. QTEMP cannot be used as the *trigger-name* schema qualifier.

If SQL names were specified, the trigger will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the trigger will be created in the schema that is specified by the qualifier. If not qualified, the trigger will be created in the same schema as the subject table.

If the trigger name is not a valid system name, or if a program with the same name already exists, the database manager will generate a system name. For information about the rules for generating a name, see "Rules for Table Name Generation" on page 1030.

NO CASCADE
NO CASCADE is allowed for compatibility with other products and is not used by DB2 for i.

BEFORE
Specifies that the trigger is a *before* trigger. The database manager executes the *triggered-action* before it applies any changes caused by an insert, delete, or update operation on the subject table. It also specifies that the *triggered-action* does not activate other triggers because the *triggered-action* of a before trigger cannot contain any updates.

BEFORE must not be specified when a *view-name* is specified. FOR EACH STATEMENT must not be specified for a BEFORE trigger.

AFTER
Specifies that the trigger is an *after* trigger. The database manager executes the *triggered-action* after it applies any changes caused by an insert, delete, or update operation on the subject table. AFTER must not be specified when *view-name* is also specified.

INSTEAD OF
Specifies that the trigger is an instead of trigger. The associated triggered action replaces the action against the subject view. Only one INSTEAD OF trigger is allowed for each kind of operation on a given subject view. The database manager executes the *triggered-action* instead of the insert, delete, or update operation on the subject view.

INSTEAD OF must not be specified when *table-name* is specified. The WHEN clause must not be specified for an INSTEAD OF trigger. FOR EACH STATEMENT must not be specified for an INSTEAD OF trigger.

trigger-event
Specifies that the triggered action associated with the trigger is to be executed whenever one of the events is applied to the subject table or view. Any combination of the events can be specified, but each event (INSERT, DELETE, UPDATE) can only be specified once.

INSERT
Specifies that the *triggered-action* associated with the trigger is to be executed whenever there is an insert operation on the subject table.

DELETE
Specifies that the *triggered-action* associated with the trigger is to be executed whenever there is a delete operation on the subject table.

A DELETE trigger cannot be added to a table with a referential constraint of ON DELETE CASCADE.

UPDATE
Specifies that the *triggered-action* associated with the trigger is to be executed whenever there is an update operation on the subject table.

An UPDATE trigger event cannot be added to a table with a referential constraint of ON DELETE SET NULL or ON DELETE SET DEFAULT.

If an explicit *column-name* list is not specified, an update operation on any column of the subject table, including columns that are subsequently added with the ALTER TABLE statement, activates the *triggered-action.*

OF *column-name*, **...**
Each *column-name* specified must be a column of the subject table, and must appear in the list only once. An update operation on any of the listed columns activates the *triggered-action*. This clause cannot be specified for an INSTEAD OF trigger.

ON *table-name*
Identifies the subject table of a BEFORE or AFTER trigger definition. The name must identify a base table that exists at the current server, but must not identify a catalog table, a table in QTEMP, or a declared temporary table.

ON *view-name*
Identifies the subject view of an INSTEAD OF trigger definition. The name must identify a view that exists at the current server, but must not identify a catalog view, or a view in QTEMP. The name must not specify a view that is defined using WITH CHECK OPTION, or a view on which a WITH CHECK OPTION view has been defined, directly or indirectly

REFERENCING
Specifies the correlation names for the transition tables and the table names for the transition tables. *Correlation-names* identify a specific row in the set of rows affected by the triggering SQL operation. *Table-identifiers* identify the complete set of affected rows.

Each row affected by the triggering SQL operation is available to the *triggered-action* by qualifying columns with *correlation-names* specified as follows:

OLD ROW AS *correlation-name*
Specifies a correlation name that identifies the values in the row prior to the triggering SQL operation. If the trigger event is insert, the value for every column in OLD ROW is the NULL value.

NEW ROW AS *correlation-name*
Specifies a correlation name which identifies the values in the row as modified by the triggering SQL operation and any SET statement in a before trigger that has already executed. If the trigger event is delete, the value for every column in NEW ROW is the NULL value.

The complete set of rows affected by the triggering SQL operation is available to the *triggered-action* by using a temporary table name specified as follows. These transition tables cannot be used for triggers that are defined for more than one *trigger-event*.

OLD TABLE AS *table-identifier*
Specifies the name of a temporary table that identifies the values in the complete set of affected rows prior to the triggering SQL operation. The OLD TABLE includes the rows that were affected by the trigger if the current activation of the trigger was caused by statements in the *SQL-trigger-body* of a trigger.

NEW TABLE AS *table-identifier*
Specifies the name of a temporary table that identifies the state of the complete set of affected rows as modified by the triggering SQL operation and by any SET statement in a before trigger that has already been executed.

Only one OLD and one NEW *correlation-name* may be specified for a trigger. Only one OLD_TABLE and one NEW_TABLE *table-identifier* may be specified for a trigger. All of the *correlation-names* and *table-identifier*s must be unique from one another.

The OLD *correlation-name* and the OLD_TABLE *table-identifier* are populated only if the triggering event is either a delete operation or an update operation. For a delete operation, the OLD *correlation-name* captures the values of the columns in the deleted row, and the OLD_TABLE *table-identifier* captures the values in the set of deleted rows. For an update operation, OLD *correlation-name* captures the values of the columns of a row before the update operation, and the OLD_TABLE *table-identifier* captures the values in the set of updated rows.

The NEW ROW *correlation-name* and the NEW TABLE *table-identifier* are valid only if the triggering event is either an INSERT operation or an UPDATE operation. For both operations, the NEW ROW *correlation-name* captures the values of the columns in the inserted or updated row, and the NEW TABLE *table-identifier* captures the values in the set of inserted or updated rows. For before triggers, the values of the updated rows include the changes from any SET statements in the *triggered-action* of before triggers.

The OLD ROW and NEW ROW *correlation-name* variables cannot be modified in an AFTER trigger or INSTEAD OF trigger.

The tables below summarizes the allowable combinations of correlation variables and transition tables.

Granularity: **FOR EACH ROW**

MODE	Activation Time	Triggering Operation	Correlation Variables Allowed	Transition Tables Allowed
DB2ROW	BEFORE	DELETE	OLD	NONE
		INSERT	NEW	
		UPDATE	OLD, NEW	
	AFTER or INSTEAD OF	DELETE	OLD	
		INSERT	NEW	
		UPDATE	OLD, NEW	
DB2SQL	BEFORE	DELETE	OLD	
		INSERT	NEW	
		UPDATE	OLD, NEW	
	AFTER or INSTEAD OF	DELETE	OLD	OLD TABLE
		INSERT	NEW	NEW TABLE
		UPDATE	OLD, NEW	OLD TABLE, NEW TABLE

Granularity: **FOR EACH STATEMENT**

MODE	Activation Time	Triggering Operation	Correlation Variables Allowed	Transition Tables Allowed
DB2SQL	AFTER or INSTEAD OF	DELETE	NONE	OLD TABLE
		INSERT		NEW TABLE
		UPDATE		OLD TABLE, NEW TABLE

A transition variable that has a character data type inherits the CCSID of the column of the subject table. During the execution of the *triggered-action*, the transition variables are treated like variables. Therefore, character conversion might occur.

The temporary transition tables are read-only. They cannot be modified.

The scope of each *correlation-name* and each *table-identifier* is the entire trigger definition.

FOR EACH ROW
Specifies that the database manager executes the *triggered-action* for each row of the subject table that the triggering operation modifies. If the triggering operation does not modify any rows, the *triggered-action* is not executed.

FOR EACH STATEMENT
Specifies that the database manager executes the *triggered-action* only once for the triggering operation. Even if the triggering operation does not modify or delete any rows, the triggered action is still executed once.

FOR EACH STATEMENT cannot be specified for a BEFORE trigger.

FOR EACH STATEMENT cannot be specified for a MODE DB2ROW trigger.

MODE DB2SQL
MODE DB2SQL is valid for AFTER triggers. MODE DB2SQL AFTER triggers are activated after all of the row operations have occurred.

MODE DB2SQL is only valid for BEFORE triggers if a REFERENCING clause is not specified and the trigger table is not referenced in the *SQL-trigger-body*. MODE DB2SQL BEFORE triggers are activated on each row operation.

MODE DB2ROW
MODE DB2ROW triggers are activated on each row operation.

MODE DB2ROW is valid for both the BEFORE and AFTER activation time.

CONCURRENT ACCESS RESOLUTION
Specifies whether the database manager should wait for data that is in the process of being updated. DEFAULT is the default.

DEFAULT
Specifies that the concurrent access resolution is not explicitly set for this trigger. The value that is in effect when the trigger program is invoked will be used.

WAIT FOR OUTCOME
Specifies that the database manager is to wait for the commit or rollback of data in the process of being updated.

USE CURRENTLY COMMITTED
Specifies that the database manager is to use the currently committed version of the data when encountering data that is in the process of being updated.

When the lock contention is between a read transaction and a delete or update transaction, the clause is applicable to scans with isolation level CS (but not for CS KEEP LOCKS).

PROGRAM NAME *external-program-name*
Specifies the unqualified name of the program to be created for the trigger. The name must be the unqualified form of an *external-program-name*. The qualifier for the program name will be the same as the implicit or explicit qualifier for *trigger-name*.

triggered-action
Specifies the action to be performed when a trigger is activated. The *triggered-action* is composed of one or more SQL statements and by an optional condition that controls whether the statements are executed.

SET OPTION-statement
Specifies the options that will be used to create the trigger. For example, to create a debuggable trigger, the following statement could be included:

```
SET OPTION DBGVIEW = *SOURCE
```

The default values for the options depend the options in effect at create time. For more information, see "SET OPTION" on page 1368.

The options CNULIGN, CNULRQD, COMPILEOPT, NAMING, and SQLCA are not allowed in the CREATE TRIGGER statement. The COMMIT option is allowed, but ignored.

The options DATFMT, DATSEP, TIMFMT, and TIMSEP cannot be used if OLD ROW or NEW ROW is specified.

WHEN (*search-condition***)**
Specifies a condition that evaluates to true, false, or unknown. The triggered SQL statements are executed only if the *search-condition* evaluates to true. If the WHEN clause is omitted, the associated SQL statements are always executed.

A WHEN clause must not be specified with an INSTEAD OF trigger.

SQL-trigger-body
Specifies a single *SQL-procedure-statement*, including a compound statement. See Chapter 7, "SQL control statements," on page 1427 for more information about defining SQL triggers.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK, SET TRANSACTION, and SET RESULT SETS statement is not allowed in the *triggered-action* of a trigger.

An UNDO handler is not allowed in a trigger.

All tables, views, aliases, distinct types, global variables, user-defined functions, and procedures referenced in the *triggered-action* must exist at the current server when the trigger is created. The table or view that an alias refers to must also exist when the trigger is created. This includes objects in library QTEMP. While objects in QTEMP can be referenced in the *triggered-action*, dropping those objects in QTEMP will not cause the trigger to be dropped.

All transition variable names are column names of the subject table. System column names of the subject table cannot be used as transition variable names.

The triggered action of a BEFORE trigger on a column of type XML can invoke the XMLVALIDATE function through a SET statement, leave values of type XML unchanged, or assign them to NULL using a SET statement.

The statements in the *triggered-action* can invoke a procedure or a user-defined function that can access a server other than the current server if the procedure or user-defined function runs in a different activation group.

Notes

Trigger ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the trigger is created exists, the *owner* of the trigger is that user profile.
- Otherwise, the *owner* of the trigger is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the trigger is the user profile or group user profile of the job executing the statement.

Trigger authority: The trigger program object authorities are:

- When SQL naming is in effect, the trigger program will be created with the public authority of *EXCLUDE, and adopt authority from the schema qualifier of the trigger-name if a user profile with that name exists. If a user profile for the schema qualifier does exist, then the owner of the trigger program will be the user profile for the schema qualifier. Note that the special authorities *ALLOBJ and *SECADM are required to create the trigger program object in the schema qualifier library if a user profile exists that has the same name as the schema qualifier, and the name is different from the authorization ID of the statement. If a user profile for the schema qualifier does not exist, then the owner of the trigger program will be the user profile or group user profile of the job executing the SQL CREATE TRIGGER statement. The group user profile will be the owner of the trigger program object, only if OWNER(*GRPPRF) was

specified on the user's profile who is executing the statement. If the owner of the trigger program is a member of a group profile, and if OWNER(*GRPPRF) was specified on the user's profile, the program will run with the adopted authority of the group profile.
- When System naming is in effect, the trigger program will be created with public authority of *EXCLUDE, and adopt authority from the user or group user profile of the job executing the SQL CREATE TRIGGER statement.

Execution authorization: The user executing the triggering SQL operation does not need authority to execute a static *triggered-SQL-statement*. A static *triggered-SQL-statement* will execute using the authority of the *owner* of the trigger.

REPLACE rules: When a trigger is recreated by REPLACE:
- Any existing comment or label is discarded.
- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved.
- The firing order of the trigger is not maintained.

Activating a trigger: Only insert, delete, or update operations can activate a trigger. A delete operation that occurs as a result of a referential constraint will not activate a trigger. Hence,
- A trigger with a DELETE trigger event cannot be added to a table with a referential constraint of ON DELETE CASCADE.
- A trigger with an UPDATE trigger event cannot be added to a table with a referential constraint of ON DELETE SET NULL or ON DELETE SET DEFAULT.

The activation of a trigger may cause *trigger cascading*. This is the result of the activation of one trigger that executes SQL statements that cause the activation of other triggers or even the same trigger again. The triggered actions may also cause updates as a result of the original modification, which may result in the activation of additional triggers. With trigger cascading, a significant chain of triggers may be activated causing significant change to the database as a result of a single delete, insert or update statement. The number of levels of cascading is limited to 200 or the maximum amount of storage allowed in the job or process, whichever comes first.

Adding triggers to enforce constraints: Adding a trigger to a table that already has rows in it will not cause the triggered actions to be executed. Thus, if the trigger is designed to enforce constraints on the data in the table, the data in the existing rows might not satisfy those constraints.

Considerations for implicitly hidden columns: In the body of a trigger, a trigger transition variable that corresponds to an implicitly hidden column can be referenced. A trigger transition table, that corresponds to a table with an implicitly hidden column, includes that column as part of the transition table.

Likewise, a trigger transition variable will exist for the column that is defined as implicitly hidden. A trigger transition variable that corresponds to an implicitly hidden column can be referenced in the body of a trigger.

Read-only views: The addition of an INSTEAD OF trigger for a view affects the read-only characteristic of the view. If a read-only view has a dependency

relationship with an INSTEAD OF trigger, the type of operation that is defined for the INSTEAD OF trigger defines whether the view is deletable, insertable, or updatable.

Transition variable values and INSTEAD OF triggers: All trigger transition variables in an INSTEAD OF trigger are nullable.

The initial values for new transition variables or new transition table columns visible in an INSTEAD OF INSERT trigger are set as follows:

- If a value is explicitly specified for a column in the INSERT statement, the corresponding new transition variable or new transition table column is that explicitly specified value.
- If a value is not explicitly specified for a column in the INSERT statement or the DEFAULT keyword is specified, the corresponding new transition variable or new transition table column is:
  - the default value of the underlying table column if the view column is updatable (without the INSTEAD OF trigger) and not based on a generated column (identity column or ROWID),
  - otherwise, the null value.

The initial values for new transition variables or new transition table columns visible in an INSTEAD OF UPDATE trigger are set as follows:

- If a value is explicitly specified for a column in the UPDATE statement, the corresponding new transition variable or new transition table column is that explicitly specified value.
- If the DEFAULT keyword is explicitly specified for a column in the UPDATE statement, the corresponding new transition variable or new transition table column is:
  - the default value of the underlying table column if the view column is updatable (without the INSTEAD OF trigger) and not based on a generated column (identity, row change timestamp, or ROWID),
  - otherwise, the null value.
- Otherwise, the corresponding new transition variable or new transition table column is the existing value of the column in the row.

Multiple triggers: Multiple triggers that have the same triggering SQL operation and activation time can be defined on a table. The triggers are activated based on the mode and the order in which they were created:

- MODE DB2ROW triggers (and native triggers created via the ADDPFTRG CL command) are fired first in the order in which they were created
- MODE DB2SQL triggers are fired next in the order in which they were created

For example, a MODE DB2ROW trigger that was created first is executed first, the MODE DB2ROW trigger that was created second is executed second.

A maximum of 300 triggers can be added to any given source table.

When a trigger is recreated using REPLACE, its position in the activation order is not maintained. It behaves as if the trigger were dropped and created again.

Adding columns to a subject table or a table referenced in the triggered action: If a column is added to the subject table after triggers have been defined, the following rules apply:

- If the trigger is an UPDATE trigger that was defined without an explicit column list, then an update to the new column will cause the activation of the trigger.
- If the SQL statements in the *triggered-action* refer to the triggering table, the new column is not accessible to the SQL statements until the trigger is recreated.
- The OLD_TABLE and NEW_TABLE transition tables will contain the new column, but the column cannot be referenced unless the trigger is recreated.

If a column is added to any table referenced by the SQL statements in the triggered-action, the new column is not accessible to the SQL statements until the trigger is recreated.

Dropping or revoking privileges on a table referenced in the triggered action: If an object such as a table, view or alias, referenced in the *triggered-action* is dropped, the access plans of the statements that reference the object will be rebuilt when the trigger is fired. If the object does not exist at that time, the corresponding INSERT, UPDATE or DELETE operation on the subject table will fail.

If a privilege that the creator of the trigger is required to have for the trigger to execute is revoked, the access plans of the statements that reference the object will be rebuilt when the trigger is fired. If the appropriate privilege does not exist at that time, the corresponding INSERT, UPDATE or DELETE operation on the subject table will fail.

Errors executing triggers: If a SIGNAL statement is executed in the *SQL-trigger-body*, an SQLCODE -438 and the SQLSTATE specified in the SIGNAL statement will be returned.

Other errors that occur during the execution of *SQL-trigger-body* statements are returned using SQLSTATE 09000 and SQLCODE -723.

Special registers in triggers: The values of the special registers are saved before a trigger is activated and are restored on return from the trigger. The values of the special registers are inherited from the triggering SQL operation.

Transaction isolation: All triggers, when they are activated, perform a SET TRANSACTION statement unless the isolation level of the application program invoking the trigger is the same as the default isolation level of the trigger program. This is necessary so that all of the operations by the trigger are performed with the same isolation level as the application program that caused the trigger to be run. The user may put their own SET TRANSACTION statements in an *SQL-control-statement* in the *SQL-trigger-body* of the trigger. If the user places a SET TRANSACTION statement within the *SQL-trigger-body* of the trigger, then the trigger will run with the isolation level specified in the SET TRANSACTION statement, instead of the isolation level of the application program that caused the trigger to be run.

If the application program that caused a trigger to be activated, is running with an isolation level other than No Commit (COMMIT(*NONE) or COMMIT(*NC)), the operations within the trigger will be run under commitment control and will not be committed or rolled back until the application commits its current unit of work. If ATOMIC is specified in the *SQL-trigger-body* of the trigger, and the application program that caused the ATOMIC trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), the operations within the trigger will not be run under commitment control. If the application that caused the trigger to be activated is running with an isolation level

of No Commit (COMMIT(*NONE) or COMMIT(*NC)), then the operations of a trigger are written to the database immediately, and cannot be rolled back.

If both system triggers defined by the Add Physical File Trigger (ADDPFTRG) CL command and SQL triggers defined by the CREATE TRIGGER statement are defined for a table, it is recommended that the system triggers perform a SET TRANSACTION statement so that they are run with the same isolation level as the original application that caused the triggers to be activated. It is also recommended that the system triggers run in the Activation Group of the calling application. If system triggers run in a separate Activation Group (ACTGRP(*NEW)), then those system triggers will not participate in the unit of the work for the calling application, nor in the unit of work for any SQL triggers. System triggers that run in a separate Activation Group are responsible for committing or rolling back any database operations they perform under commitment control. Note that SQL triggers defined by the CREATE TRIGGER statement always run in the caller's Activation Group.

If the triggering application is running with commitment control, the operations of an SQL trigger, and any cascaded SQL triggers, will be captured into a sub-unit of work. If the operations of the trigger and any cascaded triggers are successful, the operations captured in the sub-unit of work will be committed or rolled back when the triggering application commits or rolls back its current unit of work. Any system triggers that run in the same Activation Group as the caller, and perform a SET TRANSACTION to the isolation level of the caller, will also participate in the sub-unit of work. If the triggering application is running without commit control, then the operations of the SQL triggers will also be run without commitment control.

If an application that causes a trigger to be activated, is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), and it issues an INSERT, UPDATE, or DELETE statement that encounters an error during the execution of the statement, no other the system and SQL triggers will still be activated following the error for that operation. However, some number of changes will already have been performed. If the triggering application is running with commitment control, the operations of any triggers that are captured in a sub-unit of work will be rolled back when the first error is encountered, and no additional triggers will be activated for the current INSERT, UPDATE, or DELETE statement.

Performance considerations: Create the trigger under the isolation level that will most often by used by the application programs that cause the trigger to fire. The SET OPTION statement can be used to explicitly choose the isolation level.

ROW triggers (especially MODE DB2ROW triggers) perform much better than TABLE level triggers.

Considerations for implicitly hidden columns: A transition variable will exist for any column defined as implicitly hidden. In the body of a trigger, a transition variable that corresponds to an implicitly hidden column can be referenced.

Triggered actions in the catalog: At the time the trigger is created, the *triggered-action* is modified as a result of the CREATE TRIGGER statement:

- Naming mode is switched to SQL naming.
- All unqualified object references are explicitly qualified
- All implicit column lists (for example, SELECT *, INSERT with no column list, UPDATE SET ROW) are expanded to be the list of actual column names.

The modified *triggered-action* is stored in the catalog.

Renaming or moving a table referenced in the triggered action: Any table (including the subject table) referenced in a *triggered-action* can be moved or renamed. However, the *triggered-action* will continue to reference the old name or schema. An error will occur if the referenced table is not found when the *triggered-action* is executed. Hence, you should drop the trigger and then re-create the trigger so that it refers to the renamed or moved table.

Datetime considerations: If OLD ROW or NEW ROW is specified, the date or time constants and the string representation of dates and times in variables that are used in SQL statements in the *triggered-action* must have a format of ISO, EUR, JIS, USA, or must match the date and time formats specified when the table was created if it was created using DDS and the CRTPF CL command. If the DDS specifications contain multiple different date or time formats, the trigger cannot be created.

Operations that invalidate triggers: An *inoperative trigger* is a trigger that is no longer available to be activated. If a trigger becomes invalid, no INSERT, UPDATE, or DELETE operations will be allowed on the subject table or view. A trigger becomes invalid if:

- The SQL statements in the *triggered-action* reference the subject table or view, the trigger is a self-referencing trigger, and the table or view is duplicated using the system CRTDUPOBJ CL command, or
- The SQL statements in the *triggered-action* reference tables or views in the from library and the objects are not found in the new library when the table or view is duplicated using the system CRTDUPOBJ CL command, or
- The table or view is restored to a new library using the system RSTOBJ or RSTLIB CL commands, and the *triggered-action* references the subject table or subject view, the trigger is a self-referencing trigger.

An invalid trigger must first be dropped before it can be recreated by issuing a CREATE TRIGGER statement. Note that dropping and recreating a trigger will affect the activation order of a trigger if multiple triggers for the same triggering operation and activation time are defined for the subject table.

Trigger program object: When a trigger is created, SQL creates a temporary source file that will contain C source code with embedded SQL statements. A program object is then created using the CRTPGM command. The SQL options used to create the program are the options that are in effect at the time the CREATE TRIGGER statement is executed. The program is created with ACTGRP(*CALLER).

The program is created with STGMDL(*SNGLVL). If the trigger runs on behalf of an application that uses STGMDL(*TERASPACE) and also uses commitment control, the entire application will need to run under a job scoped commitment definition (STRCMTCTL CMTSCOPE(*JOB)).

The trigger will execute with the adopted authority of the *owner* of the trigger.

Examples

Example 1: Create two triggers that track the number of employees that a company manages. The triggering table is the EMPLOYEE table, and the triggers increment and decrement a column with the total number of employees in the

COMPANY_STATS table. The COMPANY_STATS table has the following properties:

```
CREATE TABLE COMPANY_STATS
  (NBEMP INTEGER,
   NBPRODUCT INTEGER,
   REVENUE DECIMAL(15,0))
```

This example uses row triggers to maintain summary data in another table.

Create the first trigger, NEW_HIRE, so that it increments the number of employees each time a new person is hired; that is, each time a new row is inserted into the EMPLOYEE table, increase the value of column NBEMP in table COMPANY_STATS by 1.

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

Create the second trigger, FORM_EMP, so that it decrements the number of employees each time an employee leaves the company; that is, each time a row is deleted from the table EMPLOYEE, decrease the value of column NBEMP in table COMPANY_STATS by 1.

```
CREATE TRIGGER FORM_EMP
  AFTER DELETE ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1;
  END
```

Example 2: Create a trigger, REORDER, that invokes user-defined function ISSUE_SHIP_REQUEST to issue a shipping request whenever a parts record is updated and the on-hand quantity for the affected part is less than 10% of its maximum stocked quantity. User-defined function ISSUE_SHIP_REQUEST orders a quantity of the part that is equal to the part's maximum stocked quantity minus its on-hand quantity. The function eliminates any duplicate requests to order the same PARTNO and sends the unique order to the appropriate supplier.

This example also shows how to define the trigger as a statement trigger instead of a row trigger. For each row in the transition table that evaluates to true for the WHERE clause, a shipping request is issued for the part.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW_TABLE AS NTABLE
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    SELECT ISSUE_SHIP_REQUEST(MAX_STOCKED - ON_HAND, PARTNO)
    FROM NTABLE
    WHERE ON_HAND < 0.10 * MAX_STOCKED;
  END
```

Example 3: Assume that table EMPLOYEE contains column SALARY. Create a trigger, SAL_ADJ, that prevents an update to an employee's salary that exceeds 20% and signals such an error. Have the error that is returned with an SQLSTATE of 75001 and a description. This example shows that the SIGNAL SQLSTATE statement is useful for restricting changes that violate business rules.

```
CREATE TRIGGER SAL_ADJ
  AFTER UPDATE OF SALARY ON EMPLOYEE
  REFERENCING OLD AS OLD_EMP
              NEW AS NEW_EMP
```

```
FOR EACH ROW MODE DB2SQL
WHEN (NEW_EMP.SALARY > (OLD_EMP.SALARY *1.20))
BEGIN ATOMIC
  SIGNAL SQLSTATE '75001'('Invalid Salary Increase - Exceeds 20%');
END
```

CREATE TYPE (Array)

The CREATE TYPE (Array) statement defines an array type at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

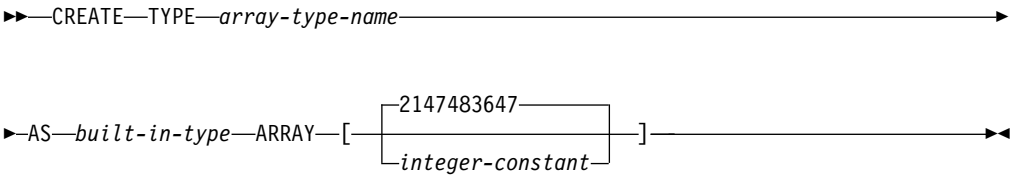
- For the SYSTYPES catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the distinct type is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

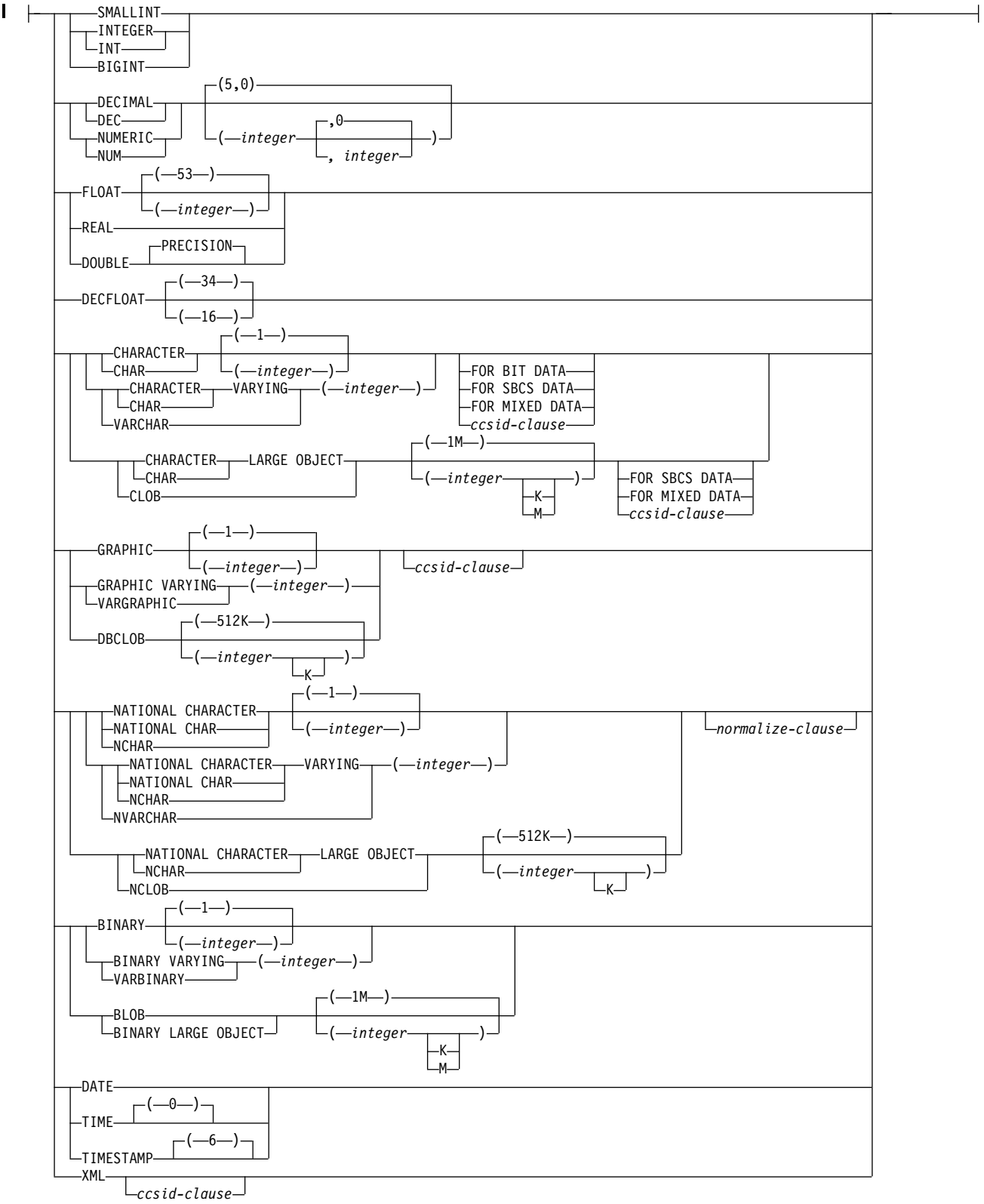
- The system authority *ADD to the user profile with that name
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

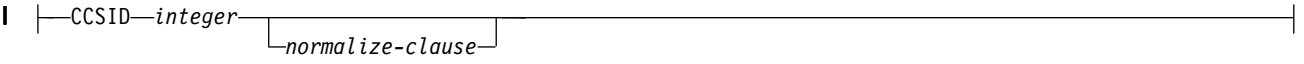
Syntax



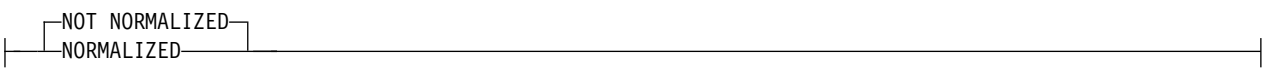
built-in-type:



ccsid-clause:



normalize-clause:



Description

array-type-name

Names the array. The name, including the implicit or explicit qualifier, must not be the same as a distinct type or array type that already exists at the current server.

If SQL names were specified, the array type will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the array type will be created in the schema that is specified by the qualifier. If not qualified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the array type will be created in the current library (*CURLIB).
- Otherwise, the array type will be created in the current schema.

If the array type name is not a valid system name, DB2 for i will generate a system name. For information about the rules for generating a name, see "Rules for Table Name Generation" on page 1030.

array-type-name must not be the name of a built-in data type, or any of the following system-reserved keywords even if you specify them as delimited identifiers.

=	<	>	>=
<=	<>	¬=	¬<
¬<	!=	!<	!>
ALL	EXTRACT	PARTITION	XMLAGG
AND	FALSE	POSITION	XMLATTRIBUTES
ANY	FOR	RID	XMLCOMMENT
ARRAY	FROM	RRN	XMLCONCAT
ARRAY_AGG	HASHED_VALUE	SELECT	XMLDOCUMENT
BETWEEN	IN	SIMILAR	XMLELEMENT
BOOLEAN	INTERVAL	SOME	XMLFOREST
CASE	IS	STRIP	XMLGROUP
CAST	LIKE	SUBSTRING	XMLNAMESPACES
CHECK	MATCH	TABLE	XMLPARSE
DATAPARTITIONNAME	NODENAME	THEN	XMLPI
DATAPARTITIONNUM	NODENUMBER	TRIM	XMLROW
DBPARTITIONNAME	NOT	TRUE	XMLSERIALIZE
DBPARTITIONNUM	NULL	TYPE	XMLTEXT
DISTINCT	ONLY	UNIQUE	XMLVALIDATE
EXCEPT	OR	UNKNOWN	XSLTRANSFORM
EXISTS	OVERLAPS	WHEN	

If a qualified *array-type-name* is specified, the schema name cannot be QSYS, QSYS2, QTEMP, or SYSIBM.

built-in-type

Specifies the built-in data type used as the data type for all the elements of the array. See "CREATE TABLE" on page 982 for a more complete description of each built-in data type.

If a specific value is not specified for the data types that have length, precision, or scale attributes, the default attributes of the data type as shown in the syntax diagram are implied.

If the array type is for a string data type, a CCSID is associated with the array type at the time the array type is created. For more information about data types, see "CREATE TABLE" on page 982.

ARRAY [*integer-constant***]**
Specifies that the array has a maximum cardinality of *integer-constant*. The value must be a positive number greater than 0. If no value is specified, the maximum integer value of 2147483647 is used. The maximum number of array elements that can be assigned in the array is the number of elements that fit in 4 gigabytes. Each varying length, LOB, and XML array element is allocated as its maximum length.

Notes

Additional generated functions: Functions are created to convert to and from the array type, but service programs are not created, so you cannot grant or revoke privileges to these functions.

Names of the generated cast functions: The unqualified name of one of the cast functions ARRAY. The name of the cast function that converts to the array type is the name of the array type. The input parameter of the cast function has the same data type as the ARRAY.

For example, assume that an array type named T_SHOESIZES is created with the following statement:

```
CREATE TYPE CLAIRE.T_SHOESIZES AS INT ARRAY[]
```

When the statement is executed, the database manager also generates the following cast functions. ARRAY converts from the array type to an array, and T_SHOESIZES converts from an array to the array type.

A generated cast function cannot be explicitly dropped. The cast functions that are generated for an array type are implicitly dropped when the array type is dropped with the DROP statement.

Array type attributes: An array type is created as a *SQLUDT object.

Array type ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the array type is created exists, the *owner* of the array type is that user profile.
- Otherwise, the *owner* of the array type is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the array type is the user profile or group user profile of the job executing the statement.

Array type authority: If SQL names are used, array types are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, array types are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the array type is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the array type.

Examples

Example 1: Create an array type named PHONENUMBERS with a maximum of 5 elements that are of the DECIMAL(10,0) data type.

```
CREATE TYPE PHONENUMBERS AS DECIMAL(10,0) ARRAY[5]
```

Example 2: Create an array type named NUMBERS in the schema GENERIC for which the maximum number of elements is not known.

```
CREATE TYPE GENERIC.NUMBERS
           AS BIGINT ARRAY[]
```

CREATE TYPE (Distinct)

The CREATE TYPE (Distinct) statement defines a distinct type at the current server. A distinct type is always sourced on one of the built-in data types.

Successful execution of the statement also generates:

- A function to cast from the distinct type to its source type
- A function to cast from the source type to its distinct type
- As appropriate, support for the use of comparison operators with the distinct type.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

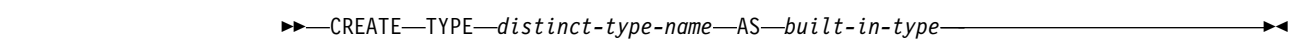
- For the SYSTYPES catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the distinct type is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

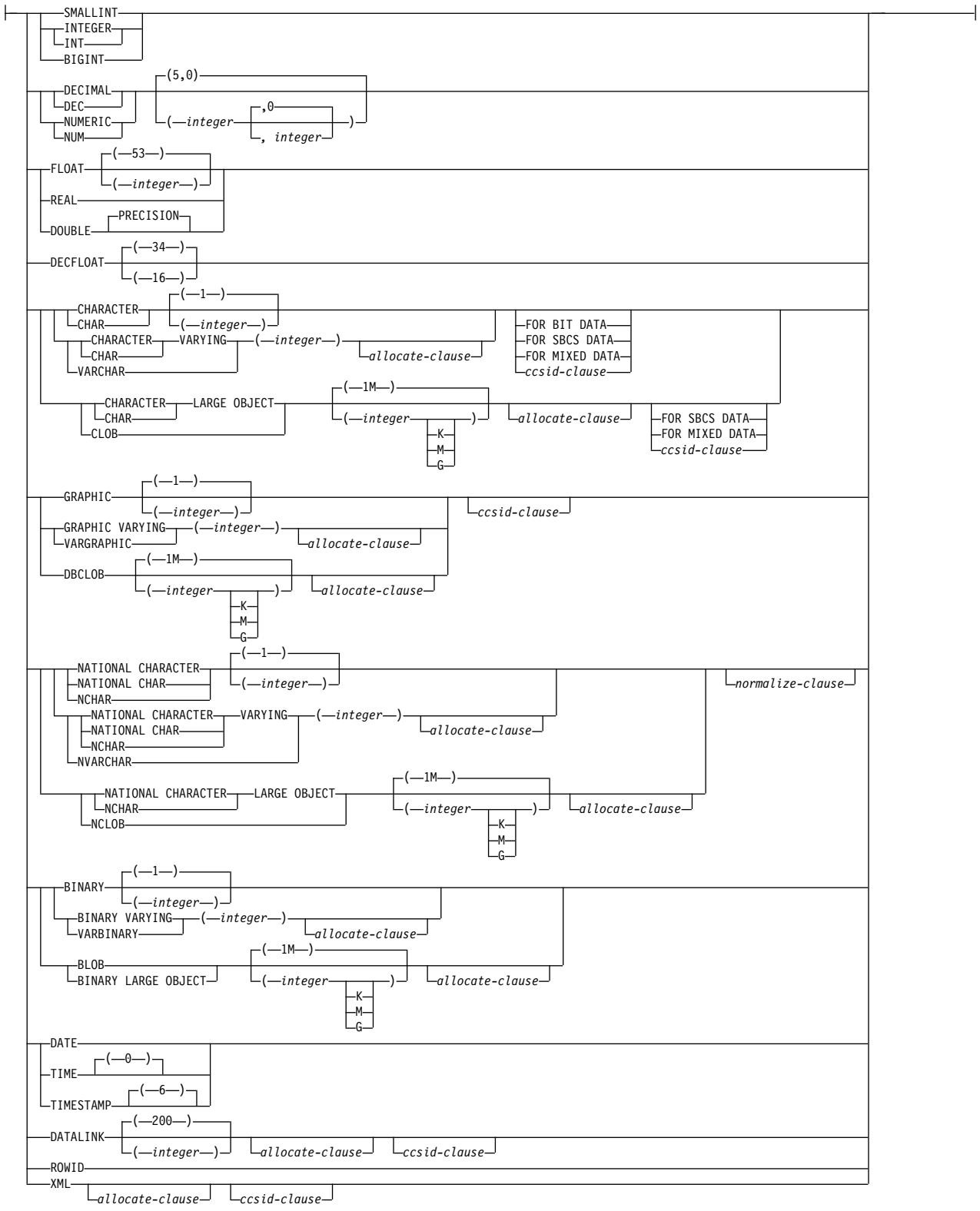
- The system authority *ADD to the user profile with that name
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

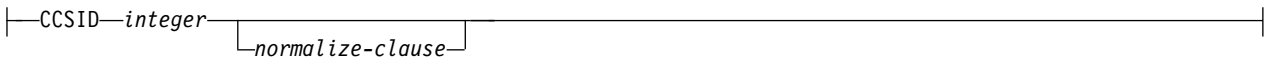
Syntax



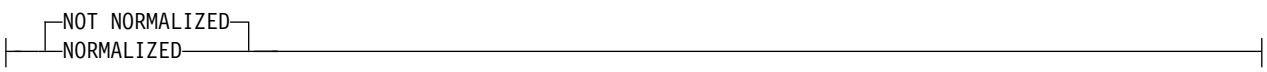
built-in-type:



ccsid-clause:



normalize-clause:



Description

distinct-type-name

Names the distinct type. The name, including the implicit or explicit qualifier, must not be the same as a distinct type or array type that already exists at the current server.

If SQL names were specified, the distinct type will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the distinct type will be created in the schema that is specified by the qualifier. If not qualified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the distinct type will be created in the current library (*CURLIB).
- Otherwise, the distinct type will be created in the current schema.

If the distinct type name is not a valid system name, DB2 for i will generate a system name. For information about the rules for generating a name, see "Rules for Table Name Generation" on page 1030.

distinct-type-name must not be the name of a built-in data type, or any of the following system-reserved keywords even if you specify them as delimited identifiers.

=	<	>	>=
<=	<>	¬=	¬<
¬<	!=	!<	!>
ALL	EXTRACT	PARTITION	XMLAGG
AND	FALSE	POSITION	XMLATTRIBUTES
ANY	FOR	RID	XMLCOMMENT
ARRAY_AGG	FROM	RRN	XMLCONCAT
BETWEEN	HASHED_VALUE	SELECT	XMLDOCUMENT
BOOLEAN	IN	SIMILAR	XMLELEMENT
CASE	INTERVAL	SOME	XMLFOREST
CAST	IS	STRIP	XMLGROUP
CHECK	LIKE	SUBSTRING	XMLNAMESPACES
DATAPARTITIONNAME	MATCH	TABLE	XMLPARSE
DATAPARTITIONNUM	NODENAME	THEN	XMLPI
DBPARTITIONNAME	NODENUMBER	TRIM	XMLROW
DBPARTITIONNUM	NOT	TRUE	XMLSERIALIZE
DISTINCT	NULL	TYPE	XMLTEXT
EXCEPT	ONLY	UNIQUE	XMLVALIDATE
EXISTS	OR	UNKNOWN	XSLTRANSFORM
	OVERLAPS	WHEN	

If a qualified *distinct-type-name* is specified, the schema name cannot be QSYS, QSYS2, QTEMP, or SYSIBM.

built-in-type

Specifies the built-in data type used as the basis for the internal representation of the distinct type. See "CREATE TABLE" on page 982 for a more complete description of each built-in data type.

For portability of applications across platforms, use the following recommended data type names:

- DOUBLE or REAL instead of FLOAT.
- DECIMAL instead of NUMERIC.

If a specific value is not specified for the data types that have length, precision, or scale attributes, the default attributes of the data type as shown in the syntax diagram are implied.

If the distinct type is sourced on a string data type, a CCSID is associated with the distinct data type at the time the distinct type is created. For more information about data types, see "CREATE TABLE" on page 982.

Notes

Additional generated functions: Besides the system-generated comparison operators described above, the following functions become available to convert to and from the source type:

- The distinct type to the source type
- The source type to the distinct type
- INTEGER to the distinct type if the source type is SMALLINT
- DOUBLE to the distinct type if the source type is REAL
- VARCHAR to the distinct type if the source type is CHAR
- VARGRAPHIC to the distinct type if the source type is GRAPHIC

These functions are created as if the following statements were executed (except that the service programs are not created, so you cannot grant or revoke privileges to these functions):

```
CREATE FUNCTION source-type-name (distinct-type-name)
  RETURNS source-type-name

CREATE FUNCTION distinct-type-name (source-type-name)
  RETURNS distinct-type-name
```

Names of the generated cast functions: Table 78 on page 1059 contains details about the generated cast functions. The unqualified name of the cast function that converts from the distinct type to the source type is the name of the source data type.

In cases in which a length, precision, or scale is specified for the source data type in the CREATE TYPE statement, the unqualified name of the cast function that converts from the distinct type to the source type is simply the name of the source data type. The data type of the value that the cast function returns includes any length, precision, or scale values that were specified for the source data type on the CREATE TYPE statement.

The name of the cast function that converts from the source type to the distinct type is the name of the distinct type. The input parameter of the cast function has the same data type as the source data type, including the length, precision, and scale.

The cast functions that are generated are created in the same schema as that of the distinct type. A function with the same name and same function signature must not already exist in the current server.

For example, assume that a distinct type named T_SHOESIZE is created with the following statement:

```
CREATE TYPE CLAIRE.T_SHOESIZE AS VARCHAR(2) WITH COMPARISONS
```

When the statement is executed, the database manager also generates the following cast functions. VARCHAR converts from the distinct type to the source type, and T_SHOESIZE converts from the source type to the distinct type.

```
FUNCTION CLAIRE.VARCHAR (CLAIRE.T_SHOESIZE) RETURNS VARCHAR(2)

FUNCTION CLAIRE.T_SHOESIZE (VARCHAR(2) RETURNS CLAIRE.T_SHOESIZE
```

Notice that function VARCHAR returns a value with a data type of VARCHAR(2) and that function T_SHOESIZE has an input parameter with a data type of VARCHAR(2).

A generated cast function cannot be explicitly dropped. The cast functions that are generated for a distinct type are implicitly dropped when the distinct type is dropped with the DROP statement.

For each built-in data type that can be the source data type for a distinct type, the following table gives the names of the generated cast functions, the data types of the input parameters, and the data types of the values that the functions returns.

Table 78. CAST Functions on Distinct Types

Source Type Name	Function Name	Parameter Type	Return Type
SMALLINT	*distinct-type-name*	SMALLINT	*distinct-type-name*
	distinct-type-name	INTEGER	*distinct-type-name*
	SMALLINT	*distinct-type-name*	SMALLINT
INTEGER	*distinct-type-name*	INTEGER	*distinct-type-name*
	INTEGER	*distinct-type-name*	INTEGER
BIGINT	*distinct-type-name*	BIGINT	*distinct-type-name*
	BIGINT	*distinct-type-name*	BIGINT
DECIMAL	*distinct-type-name*	DECIMAL(p,s)	*distinct-type-name*
	DECIMAL	*distinct-type-name*	DECIMAL(p,s)
NUMERIC	*distinct-type-name*	NUMERIC(p,s)	*distinct-type-name*
	NUMERIC	*distinct-type-name*	NUMERIC(p,s)
REAL or FLOAT(n) where n <= 24	*distinct-type-name*	REAL	*distinct-type-name*
	distinct-type-name	DOUBLE	*distinct-type-name*
	REAL	*distinct-type-name*	REAL
DOUBLE or DOUBLE PRECISION or FLOAT(n) where n > 24	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
DECFLOAT	*distinct-type-name*	DECFLOAT(n)	*distinct-type-name*
	DECFLOAT	*distinct-type-name*	DECFLOAT(n)
CHAR	*distinct-type-name*	CHAR(n)	*distinct-type-name*

Table 78. CAST Functions on Distinct Types (continued)

Source Type Name	Function Name	Parameter Type	Return Type
	CHAR	*distinct-type-name*	CHAR(n)
	distinct-type-name	VARCHAR(n)	*distinct-type-name*
VARCHAR	*distinct-type-name*	VARCHAR(n)	*distinct-type-name*
	VARCHAR	*distinct-type-name*	VARCHAR(n)
CLOB	*distinct-type-name*	CLOB(n)	*distinct-type-name*
	CLOB	*distinct-type-name*	CLOB(n)
GRAPHIC	*distinct-type-name*	GRAPHIC(n)	*distinct-type-name*
	GRAPHIC	*distinct-type-name*	GRAPHIC(n)
	distinct-type-name	VARGRAPHIC(n)	*distinct-type-name*
VARGRAPHIC	*distinct-type-name*	VARGRAPHIC(n)	*distinct-type-name*
	VARGRAPHIC	*distinct-type-name*	VARGRAPHIC(n)
DBCLOB	*distinct-type-name*	DBCLOB(n)	*distinct-type-name*
	DBCLOB	*distinct-type-name*	DBCLOB(n)
BINARY	*distinct-type-name*	BINARY(n)	*distinct-type-name*
	BINARY	*distinct-type-name*	BINARY(n)
	distinct-type-name	VARBINARY(n)	*distinct-type-name*
VARBINARY	*distinct-type-name*	VARBINARY(n)	*distinct-type-name*
	VARBINARY	*distinct-type-name*	VARBINARY(n)
BLOB	*distinct-type-name*	BLOB(n)	*distinct-type-name*
	BLOB	*distinct-type-name*	BLOB(n)
DATE	*distinct-type-name*	DATE	*distinct-type-name*
	DATE	*distinct-type-name*	DATE
TIME	*distinct-type-name*	TIME	*distinct-type-name*
	TIME	*distinct-type-name*	TIME
TIMESTAMP	*distinct-type-name*	TIMESTAMP	*distinct-type-name*
	TIMESTAMP	*distinct-type-name*	TIMESTAMP
DATALINK	*distinct-type-name*	DATALINK	*distinct-type-name*
	DATALINK	*distinct-type-name*	DATALINK
ROWID	*distinct-type-name*	ROWID	*distinct-type-name*
	ROWID	*distinct-type-name*	ROWID

NUMERIC and FLOAT are not recommended when creating a distinct type for a portable application. DECIMAL and DOUBLE should be used instead.

Built-in functions: The functions described in the above table are the only functions that are generated automatically when distinct types are defined. Consequently, none of the built-in functions (AVG, MAX, LENGTH, and so on) are automatically supported for the distinct type. A built-in function can be used on a distinct type only after a sourced user-defined function, which is based on the built-in function, has been created for the distinct type. See “Extending or overriding a built-in function:” on page 853.

The schema name of the distinct type must be included in the distinct type for successful use of these operators and cast functions in SQL statements.

Distinct type attributes: A distinct type is created as a *SQLUDT object.

Distinct type ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the distinct type is created exists, the *owner* of the distinct type is that user profile.
- Otherwise, the *owner* of the distinct type is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the distinct type is the user profile or group user profile of the job executing the statement.

Distinct type authority: If SQL names are used, distinct types are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, distinct types are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the distinct type is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the distinct type.

Syntax alternatives: The WITH COMPARISONS clause, which specifies that system-generated comparison operators are to be created for comparing two instances of the distinct type, can be specified as the last clause of the statement. Use WITH COMPARISONS only if it is required for compatibility with other products in the DB2 family.

For compatibility with previous versions of DB2:

- CREATE DISTINCT TYPE can be specified in place of CREATE TYPE.

Examples

Example 1: Create a distinct type named SHOESIZE that is sourced on the built-in INTEGER data type.

```
CREATE TYPE SHOESIZE AS INTEGER WITH COMPARISONS
```

The successful execution of this statement also generates two cast functions. Function INTEGER(SHOESIZE) returns a value with data type INTEGER, and function SHOESIZE(INTEGER) returns a value with distinct type SHOESIZE.

Example 2: Create a distinct type named MILES that is sourced on the built-in DOUBLE data type.

```
CREATE TYPE MILES
        AS DOUBLE WITH COMPARISONS
```

The successful execution of this statement also generates two cast functions. Function DOUBLE(MILES) returns a value with data type DOUBLE, and function MILES(DOUBLE) returns a value with distinct type MILES.

Example 3: Create a distinct type T_DEPARTMENT that is sourced on the built-in CHAR data type.

```
CREATE TYPE CLAIRE.T_DEPARTMENT AS CHAR(3)
   WITH COMPARISONS
```

The successful execution of this statement also generates three cast functions:

- Function CLAIRE.CHAR takes a T_DEPARTMENT as input and returns a value with data type CHAR(3).
- Function CLAIRE.T_DEPARTMENT takes a CHAR(3) as input and returns a value with distinct type T_DEPARTMENT.
- Function CLAIRE.T_DEPARTMENT takes a VARCHAR(3) as input and returns a value with distinct type T_DEPARTMENT.

CREATE VARIABLE

The CREATE VARIABLE statement defines a global variable at the application server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSVARIABLES catalog table:
  - The INSERT privilege on the table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

If a distinct type or sequence is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the distinct type or sequence identified in the statement:
  - The USAGE privilege on the distinct type or sequence, and
  - The system authority *EXECUTE on the library containing the distinct type or sequence
- Administrative authority

If a function is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the function identified in the statement:
  - The EXECUTE privilege on the function, and
  - The system authority *EXECUTE on the library containing the function
- Administrative authority

If a global variable is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement:
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

If a table or view is referenced directly or indirectly, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each table and view referenced directly or indirectly:
  - The SELECT privilege on the table or view, and

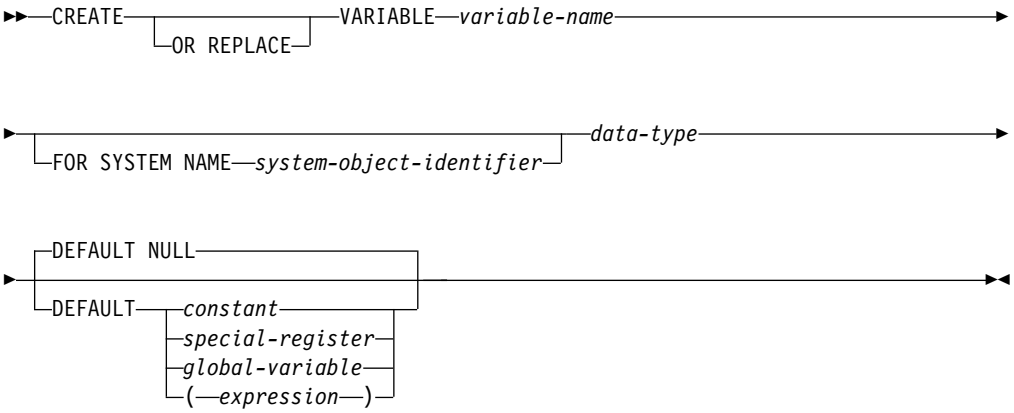
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

To replace an existing variable, the privileges held by the authorization ID of the statement must include at least one of the following:

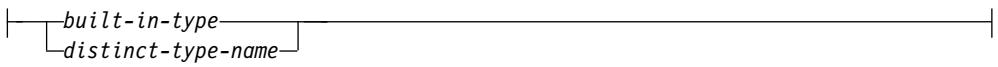
- The following system authorities:
  - The system authority of *OBJMGT on the service program for the variable
  - All authorities needed to DROP the variable
  - The system authority *READ to the SYSVARIABLES catalog table
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Sequence and Corresponding System Authorities When Checking Privileges to a Distinct Type.

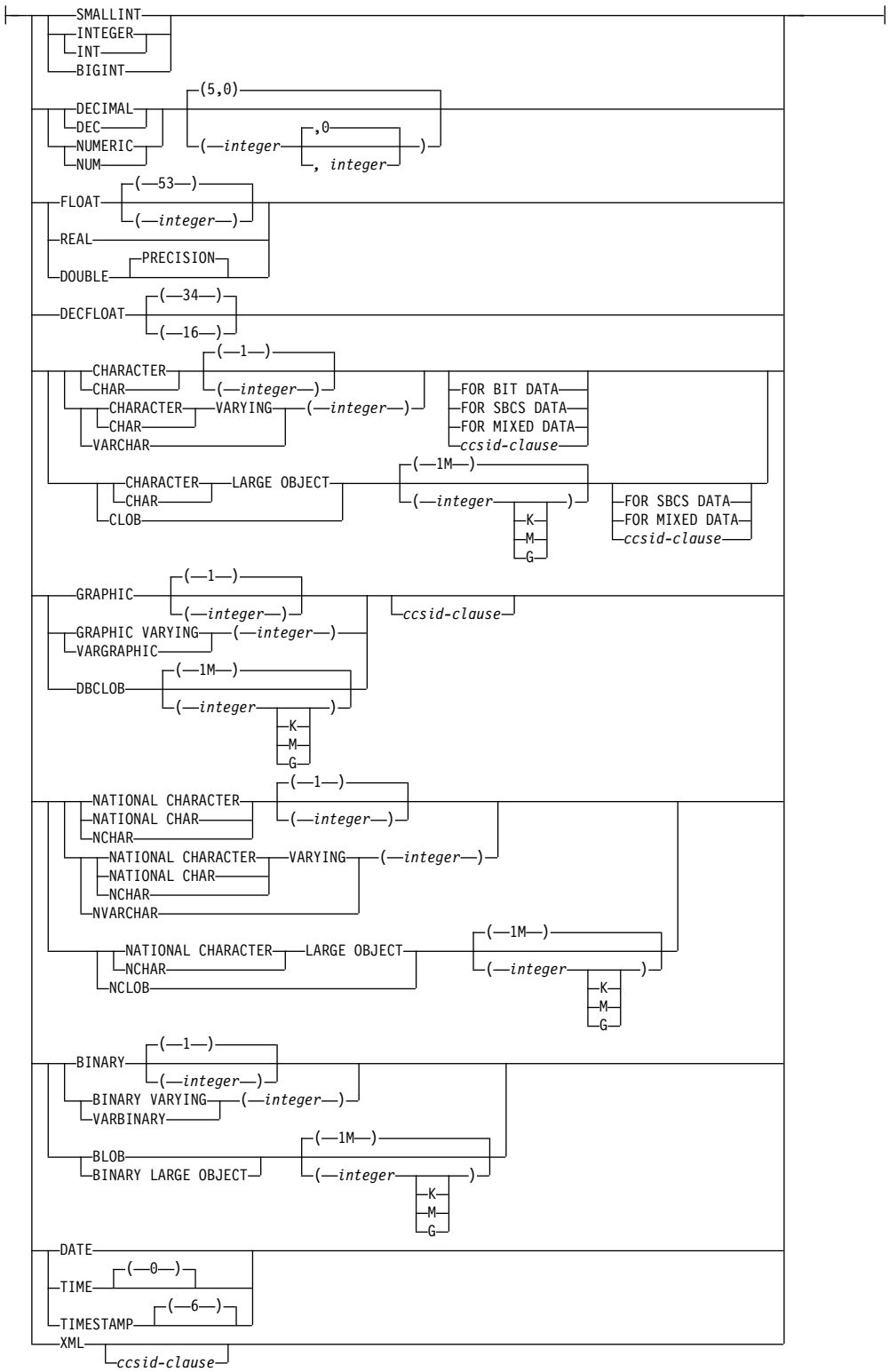
Syntax



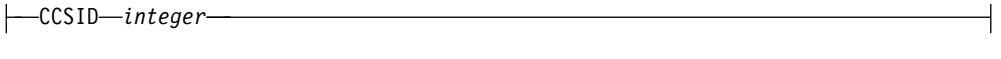
data-type:



built-in-type:



ccsid-clause:



Description

OR REPLACE
Specifies to replace the definition for the variable if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog with the exception that privileges that were granted on the variable are not affected. This option is ignored if a definition for the variable does not exist at the current server.

variable-name
Names the global variable. The name, including the implicit or explicit qualifier, must not identify a global variable that already exists at the current server. If a qualified variable name is specified, the *schema-name* cannot be QSYS2, QSYS, QTEMP, or SYSIBM.

If SQL names were specified, the variable will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the variable will be created in the schema that is specified by the qualifier. If not qualified:

- If the value of the CURRENT SCHEMA special register is *LIBL, the variable will be created in the current library (*CURLIB).
- Otherwise, the variable will be created in the current schema.

FOR SYSTEM NAME *system-object-identifier*
Identifies the *system-object-identifier* of the global variable. *system-object-identifier* must not be the same as a global variable that already exists at the current server. The *system-object-identifier* must be an unqualified system identifier.

When *system-object-identifier* is specified, *variable-name* must not be a valid system object name.

data-type
Specifies the data type or the global variable.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 982 for a more complete description of each built-in data type.

distinct-type-name
Specifies a distinct type. The length, precision, and scale of the global variable are, respectively, and length, precision, and scale of the distinct type. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path. The same limitations that apply to built-in types apply to the source type of the distinct type.

DEFAULT
Specifies a default value for the global variable. The value can be a constant, a special register, a global variable, an expression, or the keyword NULL. If a default value is not specified, the variable is initialized to the null value.

The default expression must not modify SQL data or perform an external action. The expression must be assignment compatible with the data type of the variable.

All tables, views, aliases, distinct types, sequences, global variables, and user-defined functions referenced in the default expression must exist at the current server when the global variable is created. The table or view that an alias refers to must also exist when the variable is created. This includes objects

in library QTEMP. While objects in QTEMP can be referenced in the default expression, dropping those objects in QTEMP will not cause the global variable to be dropped.

Notes

- Global variables have a session scope. This means that although they are available to all sessions that are active on the database, their value is private for each session.
- Modifications to the value of a global variable are not under transaction control. The value of the global variable is preserved when a transaction ends with either a COMMIT or a ROLLBACK statement.
- Once a global variable is instantiated for a session, changes to the global variable in another session (such as DROP or GRANT) might not affect the variable that has been instantiated.
- **Privileges to use a global variable:** An attempt to read from or to write to a global variable created by this statement requires that the authorization ID attempting this action hold the appropriate privilege on the global variable. The definer of the variable is implicitly granted all privileges on the variable.
- **Setting of the default value:** A created global variable is instantiated to its default value when it is first referenced within its given scope. Note that if a global variable is referenced in a statement, it is instantiated independently of the control flow for that statement.
- **Creating the global variable:** A global variable is created as a *SRVPGM object. If the variable name is a valid system name but a *SRVPGM already exists with that name, an error is issued. If the variable name is not a valid system name, a unique name is generated using the rules for generating system table names. For information about the rules for generating a name, see "Rules for Table Name Generation" on page 1030.

  The global variable's definition is saved in the associated service program object. If the *SRVPGM object is saved and then restored to this or another system, the catalogs are automatically updated with the definition.

  During restore of the global variable:

  – If a *SRVPGM object with the same system name exists, the *SRVPGM will be replaced.

  If a global variable and an SQL routine have the same name, naming conflicts can be avoided by creating the global variable first.
- **Using a newly created session global variable:** If a global variable is created within a session, it cannot be used by other sessions until the unit of work has been committed. However, the new global variable can be used within the session that created the variable before the unit of work commits.

Variable ownership: The *owner* of the variable is the user profile or group user profile of the job executing the statement.

Variable authority: If SQL names are used, variables are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, variables are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the variable is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the variable.

Variable instantiation authorization: When a global variable is instantiated, the DEFAULT clause is evaluated using the authority of the owner of the global variable.

REPLACE rules: When a variable is recreated by REPLACE:

- Any existing comment or label is discarded.
- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved.

Examples

Example 1: Create a global variable to indicate what printer to use for the session.

```
CREATE VARIABLE MYSCHEMA.MYJOB_PRINTER VARCHAR(30)
      DEFAULT 'Default printer'
```

Example 2: Create a global variable to indicate the department where an employee works.

```
CREATE VARIABLE SCHEMA1.GV_DEPTNO INTEGER
      DEFAULT ((SELECT DEPTNO FROM HR.EMPLOYEES
                WHERE EMPUSER = SESSION_USER))
```

Example 2: Create a global variable to indicate the security level of the current user.

```
CREATE VARIABLE SCHEMA2.GV_SECURITY_LEVEL INTEGER
      DEFAULT ( GET_SECURITY_LEVEL ( SESSION_USER))
```

CREATE VIEW

The CREATE VIEW statement creates a view on one or more tables or views at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema. For more information, see Privileges necessary to create in a schema.
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - *USE to the Create Logical File (CRTLF) CL command
  - *CHANGE to the data dictionary if the library into which the view is created is an SQL schema with a data dictionary
- Administrative authority

The privileges held by the authorization ID of the statement must also include at least one of the following:

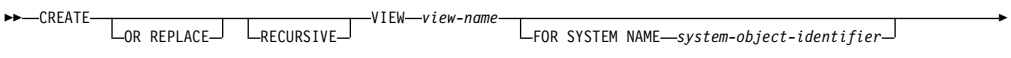
- For each table and view referenced directly through the fullselect, or indirectly through views referenced in the fullselect:
  - The SELECT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

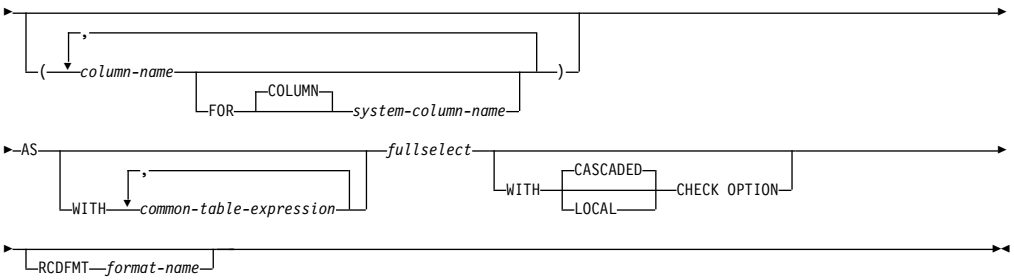
To replace an existing view, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority of *OBJMGT on the view
  - All authorities needed to DROP the view
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View and Corresponding System Authorities When Checking Privileges to a Distinct Type.

Syntax





Description

OR REPLACE

Specifies to replace the definition for the view if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog with the exception that privileges that were granted on the view are not affected. The existing object cannot be a logical file.

A definition for the view exists if:

- FOR SYSTEM NAME is specified and the *system-object-identifier* matches the *system-object-identifier* of an existing view.
- FOR SYSTEM NAME is not specified and *view-name* is a system object name that matches the *system-object-identifier* of an existing view.

If a definition for the view exists and *view-name* is not a system object name, *view-name* can be changed to provide a new name for the view.This option is ignored if a definition for the view does not exist at the current server.

RECURSIVE

Indicates that the view is potentially recursive.

If a *fullselect* of the view contains a reference to the view itself in a FROM clause, the view is a *recursive view*. Views using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The restrictions that apply to a recursive *view* are similar to those for a recursive common table expression:

- A list of *column-names* must be specified following the *view-name* unless the result columns of the fullselect are already named.
- The UNION ALL set operator must be specified.
- The first fullselect of the first union (the initialization fullselect) must not include a reference to the *view* itself in any FROM clause.
- Each fullselect that is part of the recursion cycle must not include any aggregate functions, GROUP BY clauses, or HAVING clauses.
- The FROM clauses of each fullselect can include at most one reference to the view that is part of a recursion cycle.
- The table being defined in the *common-table-expression* cannot be referenced in a subquery of a fullselect that defines the *common-table-expression*.
- LEFT OUTER JOIN and FULL OUTER JOIN are not allowed if the *common-table-expression* is the right operand. RIGHT OUTER JOIN and FULL OUTER JOIN are not allowed if the *common-table-expression* is the left operand.

If a column name of the view is referred to in the iterative fullselect, the attributes of the result columns are determined using the rules for result columns. For more information see “Rules for result data types” on page 115.

Recursive views are not allowed if the query specifies:

- a distributed table,
- a table with a read trigger,
- a table referenced directly or indirectly in the fullselect must not be a DDS-created logical file, or
- a logical file built over multiple physical file members.

view-name
Names the view. The name, including the implicit or explicit qualifier, must not be the same as an alias, file, index, table, or view that already exists at the current server.

If SQL names were specified, the view will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the view will be created in the schema that is specified by the qualifier. If not qualified and there is no default schema, the view name will be created in the same schema as the first table specified on the first FROM clause (including FROM clauses in any common table expressions or nested table expression). If no tables are referenced in the fullselect, the view will be created in the same schema as the first user defined table function. If no table or user defined table function is referenced in the fullselect, the current library (*CURLIB) will be used.

If a view name is not a valid system name and the FOR SYSTEM NAME clause is not used, DB2 for i SQL will generate a system name. For information about the rules for generating the name, see "Rules for Table Name Generation" on page 1030.

FOR SYSTEM NAME *system-object-identifier*
Identifies the *system-object-identifier* of the view. *system-object-identifier* must not be the same as a table, view, alias, or index that already exists at the current server. The *system-object-identifier* must be an unqualified system identifier.

When *system-object-identifier* is specified, *view-name* must not be a valid system object name.

(column-name, ...)
Names the columns in the view. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* and *system-column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the view inherit the names of the columns and system names of the columns of the result table of the *fullselect*.

A list of column names (and system column names) must be specified if the result table of the subselect has duplicate column names, duplicate system column names, or an unnamed column. For more information about unnamed columns, see "Names of result columns" on page 631.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. Do not use the same name for more than one column of the view or for a column-name of the view.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see "Rules for Column Name Generation" on page 1029.

AS Defines the view.

WITH *common-table-expression*
Defines a common table expression for use with the fullselect that follows. For an explanation of common table expression, see "common-table-expression" on page 683.

fullselect
Defines the view. At any time, the view consists of the rows that would result if the fullselect were executed.

fullselect must not reference variables, but may reference global variables.

The maximum number of columns allowed in a view is 8000. The column name lengths and the length of the WHERE clause also reduce this number. The maximum number of base tables allowed in the view is 256.

For an explanation of *fullselect*, see "fullselect" on page 676.

common-table-expression defines a common table expression for use with the *fullselect* that follows. For more information see "common-table-expression" on page 683.

A declared temporary table must not be referenced in the *fullselect* unless the view is created in schema QTEMP.

The ORDER BY and FETCH FIRST clauses may not be specified except within a *common-table-expression*. The OFFSET clause cannot be specified.

WITH CASCADED CHECK OPTION or WITH LOCAL CHECK OPTION
Specifies that every row that is inserted or updated through the view must conform to the definition of the view. A row that does not conform to the definition of the view is a row that cannot be retrieved using that view.

CHECK OPTION must not be specified if:

- the view is read-only
- the definition of the view includes a subquery other than a scalar fullselect in the outer select list of the view
- the definition of the view contains a non-deterministic, MODIFIES SQL DATA, or EXTERNAL ACTION function in other than the outer select list of the view
- the definition of the view contains a special register in other than the outer select list of the view
- the view references another view and that view has an INSTEAD OF trigger
- the view is recursive

If CHECK OPTION is specified for an updatable view that does not allow inserts, then the check option applies to updates only.

If CHECK OPTION is omitted, the definition of the view is not used in the checking of any insert or update operations that use the view. Some checking might still occur during insert or update operations if the view is directly or indirectly dependent on another view that includes a CHECK OPTION. Because the definition of the view is not used, rows that do not conform to the definition of the view might be inserted or updated through the view.

The difference between the two forms of the CHECK OPTION clause, CASCADED and LOCAL, is meaningful only when a view is dependent on another view. The default is CASCADED. The view upon which another view is directly or indirectly defined is an *underlying view*.

CASCADED
The WITH CASCADED CHECK OPTION on a view V is inherited by any

updatable view that is directly or indirectly dependent on V. Thus, if an updatable view is defined on V, the check option on V also applies to that view, even if WITH CHECK OPTION is not specified on that view. For example, consider the following updatable views:

```
CREATE VIEW V1 AS SELECT COL1 FROM T1 WHERE COL1 > 10

CREATE VIEW V2 AS SELECT COL1 FROM V1 WITH CHECK OPTION

CREATE VIEW V3 AS SELECT COL1 FROM V2 WHERE COL1 < 100
```

SQL statement	Description of result
INSERT INTO V1 VALUES(5)	Succeeds because V1 does not have a CHECK OPTION clause and it is not dependent on any other view that has a CHECK OPTION clause.
INSERT INTO V2 VALUES(5)	Results in an error because the inserted row does not conform to the search condition of V1 which is implicitly part of the definition of V2.
INSERT INTO V3 VALUES(5)	Results in an error because V3 is dependent on V2 which has a CHECK OPTION clause and the inserted row does not conform to the definition of V2.
INSERT INTO V3 VALUES(200)	Succeeds even though it does not conform to the definition of V3 (V3 does not have the view CHECK OPTION clause specified); it does conform to the definition of V2 (which does have the view CHECK OPTION clause specified).

LOCAL

WITH LOCAL CHECK OPTION is identical to WITH CASCADED CHECK OPTION except that it is still possible to update a row so that it no longer conforms to the definition of the view when the view is defined with the WITH LOCAL CHECK OPTION. This can only happen when the view is directly or indirectly dependent on a view that was defined without either WITH CASCADED CHECK OPTION or WITH LOCAL CHECK OPTION clauses.

WITH LOCAL CHECK OPTION specifies that the search conditions of the following underlying views are checked when a row is inserted or updated:

- views that specify WITH LOCAL CHECK OPTION
- views that specify WITH CASCADED CHECK OPTION
- all underlying views of a view that specifies WITH CASCADED CHECK OPTION

In contrast, WITH CASCADED CHECK OPTION specifies that the search conditions of all underlying views are checked when a row is inserted or updated.

The difference between CASCADED and LOCAL is best shown by example. Consider the following updatable views where x and y represent either LOCAL or CASCADED:

```
V1 defined on table T0
V2 defined on V1 WITH x CHECK OPTION
V3 defined on V2
V4 defined on V3 WITH y CHECK OPTION
V5 defined on V4
```

The following table describes which views search conditions are checked during an INSERT or UPDATE operation:

Table 79. Views whose search conditions are checked during INSERT and UPDATE

View used in INSERT or UPDATE	x = LOCAL y = LOCAL	x = CASCADED y = CASCADED	x = LOCAL y = CASCADED	x = CASCADED y = LOCAL
V1	none	none	none	none
V2	V2	V2 V1	V2	V2 V1
V3	V2	V2 V1	V2	V2 V1
V4	V4 V2	V4 V3 V2 V1	V4 V3 V2 V1	V4 V2 V1
V5	V4 V2	V4 V3 V2 V1	V4 V3 V2 V1	V4 V2 V1

RCDFMT *format-name*

An unqualified name that designates the IBM i record format name of the view. A *format-name* is a system identifier.

If a record format name is not specified, the *format-name* is the same as the *system-object-name* of the view.

Notes

View ownership: If SQL names were specified:

- If a user profile with the same name as the schema into which the view is created exists, the *owner* of the view is that user profile.
- Otherwise, the *owner* of the view is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the view is the user profile or group user profile of the job executing the statement.

View authority: If SQL names are used, views are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, views are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the view is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the view.

The owner always acquires the SELECT privilege WITH GRANT OPTION on the view and the authorization to drop the view.

The owner can also acquire the INSERT, UPDATE, and DELETE privileges on the view. If the view is not read-only, then the same privileges will be acquired on the new view as the owner has on the table or view identified in the first FROM clause of the fullselect. These privileges can be granted only if the privileges from which they are derived can also be granted.

REPLACE rules: When a view is recreated by REPLACE:

- Any existing comment or label is discarded.
- Authorized users are maintained. The object owner could change.
- Current journal auditing is preserved. However, unlike other objects, REPLACE of a view will generate a ZC (change object) journal audit entry.

- Any INSTEAD OF triggers defined for the view are dropped (any triggers that reference the view are not dropped).
- Any views and materialized query tables dependent on the view will be recreated, if possible. If it is not possible to recreate a dependent view or materialized query table, an error is returned.

Deletable views: A view is *deletable* if an INSTEAD OF trigger for the delete operation has been defined for the view, or if all of the following are true:

- The outer fullselect identifies only one base table or deletable view that is not a catalog table or view and is not in a nested table expression.
- The outer fullselect does not include a VALUES clause.
- The outer fullselect does not include a GROUP BY clause or HAVING clause.
- The outer fullselect does not include aggregate functions in the select list.
- The outer fullselect does not include a UNION, UNION ALL, EXCEPT, or INTERSECT operator.
- The outer fullselect does not include the DISTINCT clause.

Updatable views: A view is *updatable* if an INSTEAD OF trigger for the update operation has been defined for the view, or if all of the following are true:

- independent of an INSTEAD OF trigger for delete, the view is deletable
- at least one column of the view is updatable.

A column of a view is *updatable* if an INSTEAD OF trigger for the update operation has been defined for the view, or if the corresponding result column of the *subselect* is derived solely from a column of a table or an updatable column of another view (that is, it is not derived from an expression that contains an operator, scalar function, constant, or a column that itself is derived from such expressions).

Insertable views: A view is *insertable* if an INSTEAD OF trigger has been defined for the view, or if at least one column of the view is updatable.

Read-only views: A view is *read-only* if it is not deletable.

A read-only view cannot be the object of an INSERT, UPDATE, or DELETE statement.

Unqualified table names: If the CREATE VIEW statement refers to an unqualified table name, the following rules are applied to determine which table is actually being referenced:

- If the unqualified name corresponds to one or more common table expression *table-identifiers* that are specified in the *fullselect*, the name identifies the common table expression that is in the innermost scope.
- Otherwise, the name identifies a persistent table, a temporary table, or a view that is present in the default schema.

Considerations for implicitly hidden columns: It is possible that the result table of the fullselect will include a column of the base table that is defined as implicitly hidden. This can occur when the implicitly hidden column is explicitly referenced in the fullselect of the view definition. However, the corresponding column of the view does not inherit the implicitly hidden attribute. Columns of a view cannot be defined as hidden.

Collating sequence: The view is created with the collating sequence in effect at the time the CREATE VIEW statement is executed. The collating sequence of the view applies to all comparisons involving SBCS data and mixed data in the view fullselect. When the view is included in a query, an intermediate result table is generated from the view fullselect. The collating sequence in effect when the query is executed applies to any selection specified in the query.

View attributes: Views are created as nonkeyed logical files. When a view is created, the file wait time and record wait time attributes are set to the default that is specified on the WAITFILE and WAITRCD keywords of the Create Logical File (CRTLF) command.

The date and time format used for date and time result columns is ISO.

A view created over a distributed table is created on all of the systems across which the table is distributed. If a view is created over more than one distributed table, and those tables are not distributed using the same nodegroup, then the view is created only on the system that performs the CREATE VIEW statement. For more information about distributed tables, see the DB2 Multisystem topic collection.

Identity and row change timestamp columns: A column of a view is considered an identity or row change timestamp column if the element of the corresponding column in the fullselect of the view definition is the name of an identity or row change timestamp column of a table, or the name of a column of a view which directly or indirectly maps to the name of an identity or row change timestamp column of a base table. In all other cases, the columns of a view will not get the identity or row change timestamp property. For example:

- the select-list of the view definition includes multiple instances of the name of an identity column (that is, selecting the same column more than once)
- the view definition involves a join
- a column in the view definition includes an expression that refers to an identity column
- the view definition includes a UNION or INTERSECT

Examples

Example 1: Create a view named MA_PROJ over the PROJECT table that contains only those rows with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE VIEW MA_PROJ
  AS SELECT * FROM PROJECT
       WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 2: Create a view as in example 1, but select only the columns for project number (PROJNO), project name (PROJNAME) and employee in charge of the project (RESPEMP).

```
CREATE VIEW MA_PROJ
  AS SELECT PROJNO, PROJNAME, RESPEMP FROM PROJECT
       WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 3: Create a view as in example 2, but, in the view, call the column for the employee in charge of the project IN_CHARGE.

```
CREATE VIEW MA_PROJ (PROJNO, PROJNAME, IN_CHARGE)
  AS SELECT PROJNO, PROJNAME, RESPEMP FROM PROJECT
       WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Note: Even though you are changing only one of the column names, the names of all three columns in the view must be listed in the parentheses that follow MA_PROJ.

Example 4: Create a view named PRJ_LEADER that contains the first four columns (PROJNO, PROJNAME, DEPTNO, RESPEMP) from the PROJECT table together with the last name (LASTNAME) of the person who is responsible for the project (RESPEMP). Obtain the name from the EMPLOYEE table by matching EMPNO in EMPLOYEE to RESEMP in PROJECT.

```
CREATE VIEW PRJ_LEADER
  AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME
       FROM PROJECT, EMPLOYEE
       WHERE RESPEMP = EMPNO
```

Example 5: Create a view as in example 4, but in addition to the columns PROJNO, PROJNAME, DEPTNO, RESEMP and LASTNAME, show the total pay (SALARY + BONUS +COMM) of the employee who is responsible. Also select only those projects with mean staffing (PRSTAFF) greater than one.

```
CREATE VIEW PRJ_LEADER (PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, TOTAL_PAY)
  AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, SALARY+BONUS+COMM
       FROM PROJECT, EMPLOYEE
       WHERE RESPEMP = EMPNO AND PRSTAFF > 1
```

Example 6: Create a recursive view that returns a similar result as a common table expression, see "Example 1: Single level explosion" on page 687.

```
CREATE RECURSIVE VIEW RPL (PART, SUBPART, QUANTITY) AS
        SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
           FROM PARTLIST ROOT
           WHERE ROOT.PART = '01'
       UNION ALL
        SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
           FROM RPL PARENT, PARTLIST CHILD
           WHERE PARENT.SUBPART = CHILD.PART

SELECT DISTINCT *
FROM RPL
ORDER BY PART, SUBPART, QUANTITY
```

DEALLOCATE DESCRIPTOR

The DEALLOCATE DESCRIPTOR statement deallocates an SQL descriptor.

Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It cannot be issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

Authorization

None required.

Syntax

Description

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation. The descriptor known in this local scope is deallocated.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session. The descriptor known to any program that executes using the same database connection is deallocated.

SQL-descriptor-name
Names the descriptor to deallocate. The name must identify a descriptor that already exists with the specified scope.

Notes

Descriptor persistence: Local and global descriptors are also implicitly deallocated. For more information, see Descriptor persistence

Examples

Deallocate a descriptor called 'NEWDA'.

```
EXEC SQL DEALLOCATE DESCRIPTOR 'NEWDA'
```

DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java.

Authorization

No authorization is required to use this statement. However to use OPEN or FETCH for the cursor, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the SELECT statement of the cursor:
  - The SELECT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

The SELECT statement of the cursor is one of the following:

- The prepared *select-statement* identified by the *statement-name*.
- The specified *select-statement*.

If ***statement-name*** is specified:

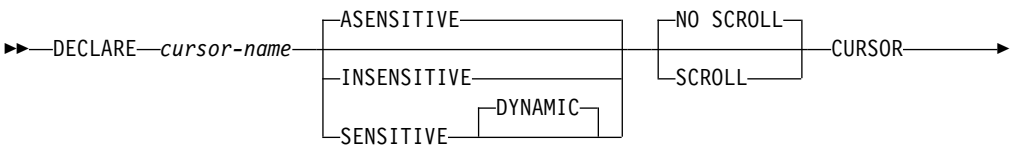
- The authorization ID of the statement is the run-time authorization ID unless DYNUSRPRF(*OWNER) was specified on the CRTSQLxxx command when the program was created. For more information, see "Authorization IDs and authorization names" on page 68.
- The authorization check is performed when the *select-statement* is prepared unless DLYPRP(*YES) is specified on the CRTSQLxxx command.
- The authorization check is performed when the cursor is opened for programs compiled with the DLYPRP(*YES) parameter.

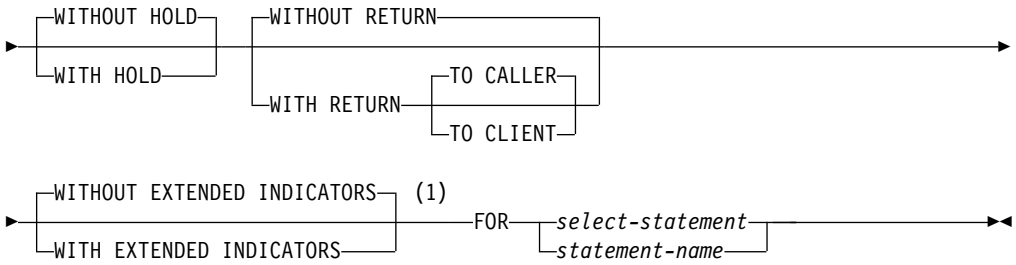
If the ***select-statement*** is specified:

- If USRPRF(*OWNER) or USRPRF(*NAMING) with SQL naming was specified on the CRTSQLxxx command, the authorization ID of the statement is the owner of the SQL program or package.
- If USRPRF(*USER) or USRPRF(*NAMING) with system naming was specified on the CRTSQLxxx command, the authorization ID of the statement is the run-time authorization ID.
- In REXX, the authorization ID of the statement is the run-time authorization ID.
- The authorization check is performed when the cursor is opened.

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

Syntax





Notes:

1 The HOLD, RETURN, and EXTENDED INDICATORS clauses can be specified in any order.

Description

cursor-name
Names a cursor. The name must not be the same as the name of another cursor declared in your source program.

ASENSITIVE, SENSITIVE, or INSENSITIVE
Specifies whether the cursor is asensitive, sensitive, or insensitive to changes. If *statement-name* is specified, the default is the corresponding prepare attribute of the statement. Otherwise, ASENSITIVE is the default.

ASENSITIVE
Specifies that the cursor may behave as SENSITIVE or INSENSITIVE depending on how the *select-statement* is optimized.

SENSITIVE
Specifies that changes made to the database after the cursor is opened are visible in the result table. The cursor has some level of sensitivity to any updates or deletes made to the rows underlying its result table after the cursor is opened. The cursor is always sensitive to positioned updates or deletes using the same cursor. Additionally, the cursor can have sensitivity to changes made outside this cursor. If the database manager cannot make changes visible to the cursor, then an error is returned. The database manager cannot make changes visible to the cursor when the cursor implicitly becomes read-only. (See Result table of a cursor.) If SENSITIVE is specified, the SELECT statement cannot contain a *data-change-table-reference*.

INSENSITIVE
Specifies that once the cursor is opened, it does not have sensitivity to inserts, updates, or deletes performed by this or any other activation group. If INSENSITIVE is specified, the cursor is read-only and a temporary result is created when the cursor is opened. In addition, the SELECT statement cannot contain a UPDATE clause and the application must allow a copy of the data (ALWCPYDTA(*OPTIMIZE) or ALWCPYDTA(*YES)).

NO SCROLL or SCROLL
Specifies whether the cursor is scrollable or not scrollable. If *statement-name* is specified, the default is the corresponding prepare attribute of the statement. Otherwise, NO SCROLL is the default.

NO SCROLL
Specifies that the cursor is not scrollable.

SCROLL
Specifies that the cursor is scrollable. The cursor may or may not have immediate sensitivity to inserts, updates, and deletes done by other activation groups.

WITHOUT HOLD or WITH HOLD
Specifies whether the cursor should be prevented from being closed as a consequence of a commit operation. If *statement-name* is specified, the default is the corresponding prepare attribute of the statement. Otherwise, WITHOUT HOLD is the default.

WITHOUT HOLD
Does not prevent the cursor from being closed as a consequence of a commit operation.

WITH HOLD
Prevents the cursor from being closed as a consequence of a commit operation. A cursor declared using the WITH HOLD clause is implicitly closed at commit time only if the connection associated with the cursor is ended during the commit operation.

When WITH HOLD is specified, a commit operation commits all the changes in the current unit of work, and releases all locks except those that are required to maintain the cursor position. Afterward, a FETCH statement is required before a Positioned UPDATE or DELETE statement can be executed.

All cursors are implicitly closed by a CONNECT (Type 1) or rollback operation. All cursors associated with a connection are implicitly closed by a disconnect of the connection. A cursor is also implicitly closed by a commit operation if WITH HOLD is not specified, or if the connection associated with the cursor is in the release-pending state.

If a cursor is closed before the commit operation, the effect is the same as if the cursor was declared without the WITH HOLD option.

WITHOUT RETURN or WITH RETURN
Specifies that the result table of the cursor is intended to be used as a procedure result set. If *statement-name* is specified, the default is the corresponding prepare attribute of the statement. Otherwise, WITHOUT RETURN is the default.

WITHOUT RETURN
Specifies that the result table of the cursor is not intended to be used as a procedure result set.

WITH RETURN
Specifies that the result table of the cursor is intended to be used as a procedure result set. If the DECLARE CURSOR statement is not contained within the source code for a procedure, the clause is ignored.

For SQL procedures, result sets are only returned if a DYNAMIC RESULT SETS clause with a nonzero maximum number of result sets is specified on the procedure definition.

- Cursors defined by using the WITH RETURN clause that are still open when the procedure ends define the result sets for the procedure. All other open cursors are closed when the procedure ends, provided the procedure was not created with CLOSQLCSR(*ENDACTGRP).

- If no cursors in the stored procedure are defined by using the WITH RETURN or WITHOUT RETURN clause, then any cursor that is open when the stored procedure ends potentially becomes a result set cursor.
- See the DYNAMIC RESULT SETS clause in "CREATE PROCEDURE (SQL)" on page 955 for further considerations that determine a procedure's result sets.

For external procedures:

- Any cursors that are defined by using the WITH RETURN clause (or identified as a result set cursor in a SET RESULT SETS statement) and that are still open when the procedure ends define the potential result sets for the procedure, provided the procedure was not created with CLOSQLCSR(*ENDACTGRP). All other open cursors remain open.
- If no cursors in the stored procedure are defined by using the WITH RETURN or WITHOUT RETURN clause, and no cursors are identified as a result set cursor in a SET RESULT SETS statement, then any cursor that is open when the stored procedure ends potentially becomes a result set cursor.
- See the DYNAMIC RESULT SETS clause in "CREATE PROCEDURE (External)" on page 939 for further considerations that determine a procedure's result sets.

For non-scrollable cursors, the result set consists of all rows from the current cursor position to the end of the result table. For scrollable cursors, the result set consists of all rows of the result table.

TO CALLER
Specifies that the cursor can return a result set to the caller of the procedure. For example, if the caller is a client application, the result set is returned to the client application.

TO CLIENT
Specifies that the cursor can return a result set to the client application. This cursor is invisible to any intermediate nested procedures. If a function or trigger called the procedure either directly or indirectly, result sets cannot be returned to the client and the cursor will be closed after the procedure finishes.

TO CLIENT may be necessary if the result set is returned from an ILE program with multiple modules.

WITHOUT EXTENDED INDICATORS or WITH EXTENDED INDICATORS
Specifies whether extended indicators are enabled. If *statement-name* is specified, the default is the corresponding prepare attribute of the statement. Otherwise, the default is the attribute specified on the containing program or service program.

WITHOUT EXTENDED INDICATORS
Specifies that extended indicator variables are not enabled, and only updatable columns are allowed in the implicit or explicit UPDATE clause or the *select-statement*.

WITH EXTENDED INDICATORS
Specifies that extended indicator variables are enabled, and non-updatable columns are allowed in the implicit or explicit UPDATE clause of the *select-statement*.

select-statement
Specifies the SELECT statement of the cursor. See "select-statement" on page 682 for more information.

The *select-statement* must not include parameter markers (except for REXX), but can include references to variables. In host languages, other than REXX, the declarations of the host variables must precede the DECLARE CURSOR statement in the source program. In REXX, parameter markers must be used in place of variables and the statement must be prepared.

statement-name
The SELECT statement of the cursor is the prepared *select-statement* identified by the *statement-name* when the cursor is opened. The *statement-name* must not be identical to a *statement-name* specified in another DECLARE CURSOR statement of the source program. See "PREPARE" on page 1293 for an explanation of prepared statements.

Notes

Placement of DECLARE CURSOR: The DECLARE CURSOR statement must precede all statements that explicitly reference the cursor by name, except in C and PL/I.

Result table of a cursor: A cursor in the open state designates a *result table* and a position relative to the rows of that table. The table is the result table specified by the SELECT statement of the cursor.

A cursor is *deletable* if all of the following are true:

- The outer fullselect identifies only one base table or deletable view that is not a catalog table or view and is not in a nested table expression.
- The outer fullselect does not include a VALUES clause.
- The outer fullselect does not include a GROUP BY clause or HAVING clause.
- The outer fullselect does not include aggregate functions in the select list.
- The outer fullselect does not include a UNION, UNION ALL, EXCEPT, or INTERSECT operator.
- The *select-clause* of the outer fullselect does not include the DISTINCT clause.
- The outer fullselect does not include a *data-change-table-reference* in the FROM clause
- The *select-statement* does not contain an ORDER BY clause and does not contain the UPDATE clause and SENSITIVE is not specified in the DECLARE CURSOR statement
- The *select-statement* does not include a FOR READ ONLY clause.
- The result of the outer fullselect does not make use of a temporary table.
- The *select-statement* does not include the SCROLL keyword, or the SENSITIVE keyword or UPDATE clause is also specified.
- The select list does not include a DATALINK column unless a UPDATE clause is specified.

A result column in the select list of the outer fullselect associated with a cursor is *updatable* if all of the following are true:

- The cursor is deletable.
- The result column is derived solely from a column of a table or an updatable column of a view. That is, at least one result column must not be derived from

an expression that contains an operator, scalar function, constant, or a column that itself is derived from such expressions.

A cursor is *read-only* if it is not deletable.

If the UPDATE clause is omitted, only the columns in the SELECT clause of the subselect that can be updated can be changed.

If UPDATE is specified without a list of column names, then the list of columns that can appear as targets in the assignment clause of subsequent positioned UPDATE statements identifying this cursor is determined as follows:

- If WITH EXTENDED INDICATORS is specified, all the columns of the table or view identified in the first FROM clause of the fullselect.
- Otherwise, only the updatable columns of the table or view identified in the first FROM clause of the fullselect.

If UPDATE is specified with a list of column names, only the columns specified in the list of column names can be appear as targets in the assignment clause in subsequent positioned UPDATE statements identifying this cursor.

Scope of a cursor: The scope of *cursor-name* is the source program in which it is defined; that is, the program submitted to the precompiler. Thus, a cursor can only be referenced by statements that are precompiled with the cursor declaration. For example, a program called from another separately compiled program cannot use a cursor that was opened by the calling program.

The scope of cursor-name is also limited to the thread in which the program that contains the cursor is running. For example, if the same program is running in two separate threads in the same job, the second thread cannot use a cursor that was opened by the first thread.

A cursor can only be referred to in the same instance of the program in the program stack unless CLOSQLCSR(*ENDJOB), CLOSQLCSR(*ENDSQL), or CLOSQLCSR(*ENDACTGRP) is specified on the CRTSQLxxx commands.

- If CLOSQLCSR(*ENDJOB) is specified, the cursor can be referred to by any instance of the program on the program stack.
- If CLOSQLCSR(*ENDSQL) is specified, the cursor can be referred to by any instance of the program on the program stack until the last SQL program on the program stack ends.
- If CLOSQLCSR(*ENDACTGRP) is specified, the cursor can be referred to by all instances of the module in the activation group until the activation group ends.

Although the scope of a cursor is the program in which it is declared, each package created from the program includes a separate instance of the cursor and more than one cursor can exist at run time. For example, assume a program using CONNECT (Type 2) statements connects to location X and location Y in the following sequence:

```
EXEC SQL DECLARE C CURSOR FOR...
EXEC SQL CONNECT TO X;
EXEC SQL OPEN C;
EXEC SQL FETCH C INTO...
EXEC SQL CONNECT TO Y;
EXEC SQL OPEN C;
EXEC SQL FETCH C INTO...
```

The second OPEN C statement does not cause an error because it refers to a different instance of cursor C.

A SELECT statement is evaluated at the time the cursor is opened. If the same cursor is opened, closed, and then opened again, the results may be different. If the SELECT statement of a cursor contains CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, all references to these special registers will yield the same respective datetime value on each FETCH. The value is determined when the cursor is opened. Multiple cursors using the same SELECT statement can be opened concurrently. They are each considered independent activities.

Using sequence expressions: For information regarding using NEXT VALUE and PREVIOUS VALUE expressions with a cursor, see Using sequence expressions with a cursor.

Blocking of data: For more efficient processing of data, the database manager can block data for read-only cursors. If a cursor is not going to be used in a Positioned UPDATE or DELETE statement, it should be declared as FOR READ ONLY.

Usage in REXX: If variables are used on the DECLARE CURSOR statement within a REXX procedure, then the DECLARE CURSOR must be the object of a PREPARE and EXECUTE.

Temporary results: Certain *select-statements* may be implemented as temporary result tables.

- A temporary result table is created when:
  - INSENSITIVE is specified
  - The ORDER BY and GROUP BY clauses specify different columns or columns in a different order.
  - The ORDER BY and GROUP BY clauses include a user-defined function or one of the following scalar functions: DLVALUE, DLURLPATH, DLURLPATHONLY, DLURLSERVER, DLURLSCHEME, or DLURLCOMPLETE for DataLinks with an attribute of FILE LINK CONTROL and READ PERMISSION DB.
  - The UNION, EXCEPT, INTERSECT, or DISTINCT clauses are specified.
  - The ORDER BY or GROUP BY clauses specify columns which are not all from the same table.
  - A logical file defined by the JOINDFT data definition specifications (DDS) keyword is joined to another file.
  - A logical file that is based on multiple database file members is specified.
  - The CURRENT or RELATIVE scroll options are specified on the FETCH statement when the select statement of the DECLARE CURSOR contains a GROUP BY clause.
  - The FETCH FIRST n ROWS ONLY clause is specified.
- Queries that include a subquery where:
  - The outermost query does not provide correlated values to any inner subselects.
  - No IN, = ANY, = SOME, or <> ALL subqueries are referenced by the outermost query.

Cursor sensitivity: The ALWCPYDTA precompile option is ignored for DYNAMIC SCROLL cursors. If sensitivity to inserts, updates, and deletes must be maintained, a temporary copy of the data is never made unless a temporary result is required to implement the query.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- DYNAMIC SCROLL is a synonym for SENSITIVE DYNAMIC SCROLL

Examples

Example 1: Declare C1 as the cursor of a query to retrieve data from the table DEPARTMENT. The query itself appears in the DECLARE CURSOR statement.

```
EXEC SQL DECLARE C1 CURSOR FOR
           SELECT DEPTNO, DEPTNAME, MGRNO
             FROM DEPARTMENT
             WHERE ADMRDEPT = 'A00';
```

Example 2: Declare C1 as the cursor of a query to retrieve data from the table DEPARTMENT. Assume that the data will be updated later with a searched update and should be locked when the query executes. The query itself appears in the DECLARE CURSOR statement.

```
EXEC SQL DECLARE C1 CURSOR FOR
           SELECT DEPTNO, DEPTNAME, MGRNO
             FROM DEPARTMENT
             WHERE ADMRDEPT = 'A00'
             FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS;
```

Example 3: Declare C2 as the cursor for a statement named STMT2.

```
EXEC SQL DECLARE C2 CURSOR FOR STMT2;
```

Example 4: Declare C3 as the cursor for a query to be used in positioned updates of the table EMPLOYEE. Allow the completed updates to be committed from time to time without closing the cursor.

```
EXEC SQL DECLARE C3 CURSOR WITH HOLD FOR
           SELECT *
             FROM EMPLOYEE
             FOR UPDATE OF WORKDEPT, PHONENO, JOB, EDLEVEL, SALARY;
```

Instead of explicitly specifying the columns to be updated, an UPDATE clause could have been used without naming the columns. This would allow all the updatable columns of the table to be updated. Since this cursor is updatable, it can also be used to delete rows from the table.

Example 5: In a C program, use the cursor C1 to fetch the values for a given project (PROJNO) from the first four columns of the EMPPROJACT table a row at a time and put them into the following host variables: EMP(CHAR(6)), PRJ(CHAR(6)), ACT(SMALLINT) and TIM(DECIMAL(5,2)). Obtain the value of the project to search for from the host variable SEARCH_PRJ (CHAR(6)). Dynamically prepare the *select-statement* to allow the project to search by to be specified when the program is executed.

```
void main ()
  {
    EXEC SQL  BEGIN DECLARE SECTION;
    char           EMP[7];
    char           PRJ[7];
```

```
    char           SEARCH_PRJ[7];
    short          ACT;
    double         TIM;
    char           SELECT_STMT[201];
    EXEC SQL  END DECLARE SECTION;
    EXEC SQL  INCLUDE SQLCA;

    strcpy(SELECT_STMT, "SELECT EMPNO, PROJNO, ACTNO, EMPTIME \
                  FROM EMPPROJACT \
                  WHERE PROJNO = ?");
    .
    .
    .
    EXEC SQL PREPARE SELECT_PRJ FROM :SELECT_STMT;

    EXEC SQL DECLARE C1 CURSOR FOR SELECT_PRJ;

/* Obtain the value for SEARCH_PRJ from the user.     */
    .
    .
    .
    EXEC SQL OPEN C1 USING :SEARCH_PRJ;

    EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM;

    if (strcmp(SQLSTATE, "02000", 5) )
      {
        data_not_found();
      }
    else
      {
        while (strcmp(SQLSTATE, "00", 2) || strcmp(SQLSTATE, "01", 2) )
          {
            EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM;
          }
      }

    EXEC SQL  CLOSE C1;
    .
    .
    .
  }
```

Example 6: The DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT. C1 is an updatable, scrollable cursor.

```
EXEC SQL DECLARE C1 SENSITIVE SCROLL CURSOR FOR
    SELECT DEPTNO, DEPTNAME, MGRNO
    FROM TDEPT
    WHERE ADMRDEPT = 'A00';
```

Example 7: Declare a cursor in order to fetch values from four columns and assign the values to variables using the Serializable (RR) isolation level:

```
DECLARE CURSOR1 CURSOR FOR
  SELECT COL1, COL2, COL3, COL4
  FROM TBLNAME WHERE COL1 = :varname
  WITH RR
```

Example 8: Assume that the EMPLOYEE table has been altered to add a generated column, WEEKLYPAY, that calculates the weekly pay based on the yearly salary. Declare a cursor to retrieve the system generated column value from a row to be inserted.

```
DECLARE C2 CURSOR FOR
  SELECT E.WEEKLYPAY
  FROM FINAL TABLE
```

```
(INSERT INTO EMPLOYEE
    (EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL, SALARY)
    VALUES('000420', 'Peter', 'U', 'Bender', 16, 31842)) AS E;
```

DECLARE GLOBAL TEMPORARY TABLE

The DECLARE GLOBAL TEMPORARY TABLE statement defines a declared temporary table for the current application process. The declared temporary table description does not appear in the system catalog. It is not persistent and cannot be shared with other application processes. Each application process that defines a declared temporary table of the same name has its own unique description of the temporary table. When the application process ends, the temporary table is dropped.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

If the LIKE or AS *select-statement* clause is specified, the privileges held by the authorization ID of the statement must include at least one of the following on any table or view specified in the LIKE clause or *as-result-table* clause:

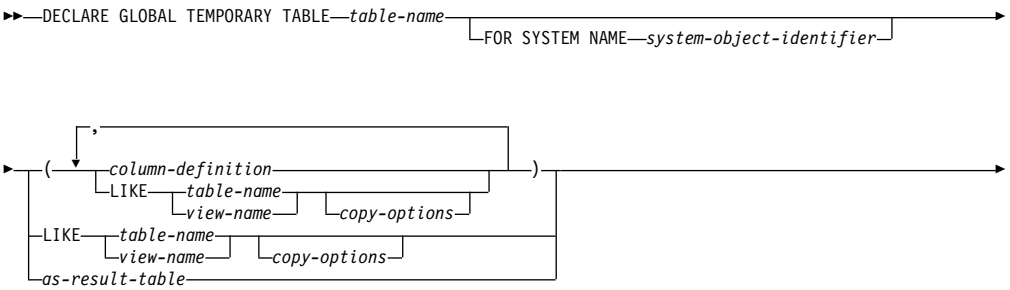
- The SELECT privilege for the table or view
- Ownership of the table or view
- Administrative authority

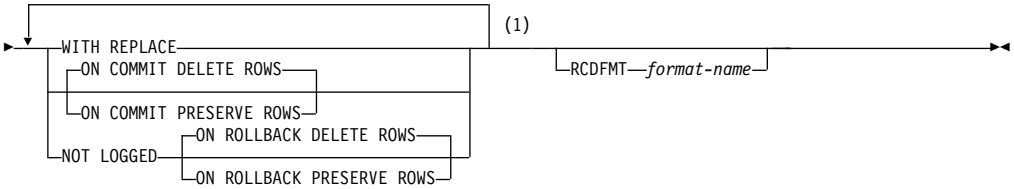
If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
  - The USAGE privilege on the distinct type, and
  - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

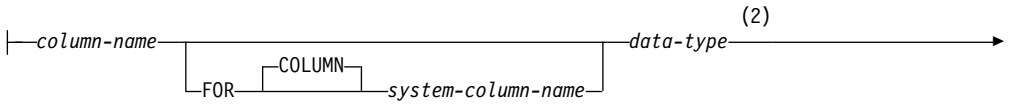
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View and Corresponding System Authorities When Checking Privileges to a Distinct Type.

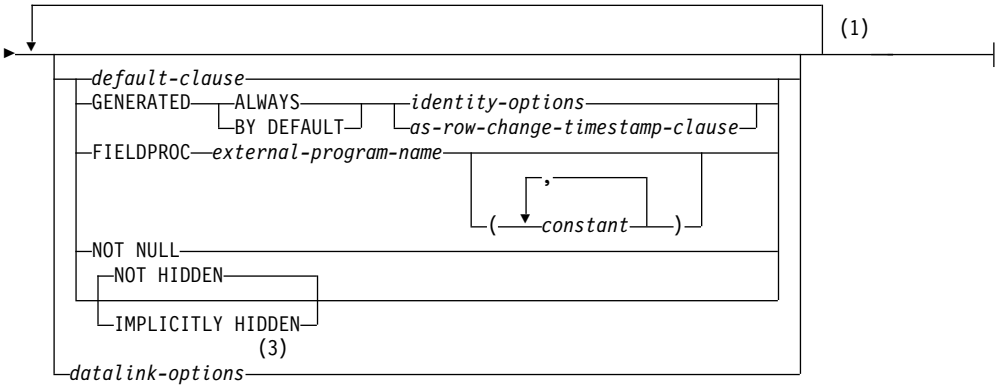
Syntax





column-definition:





data-type:

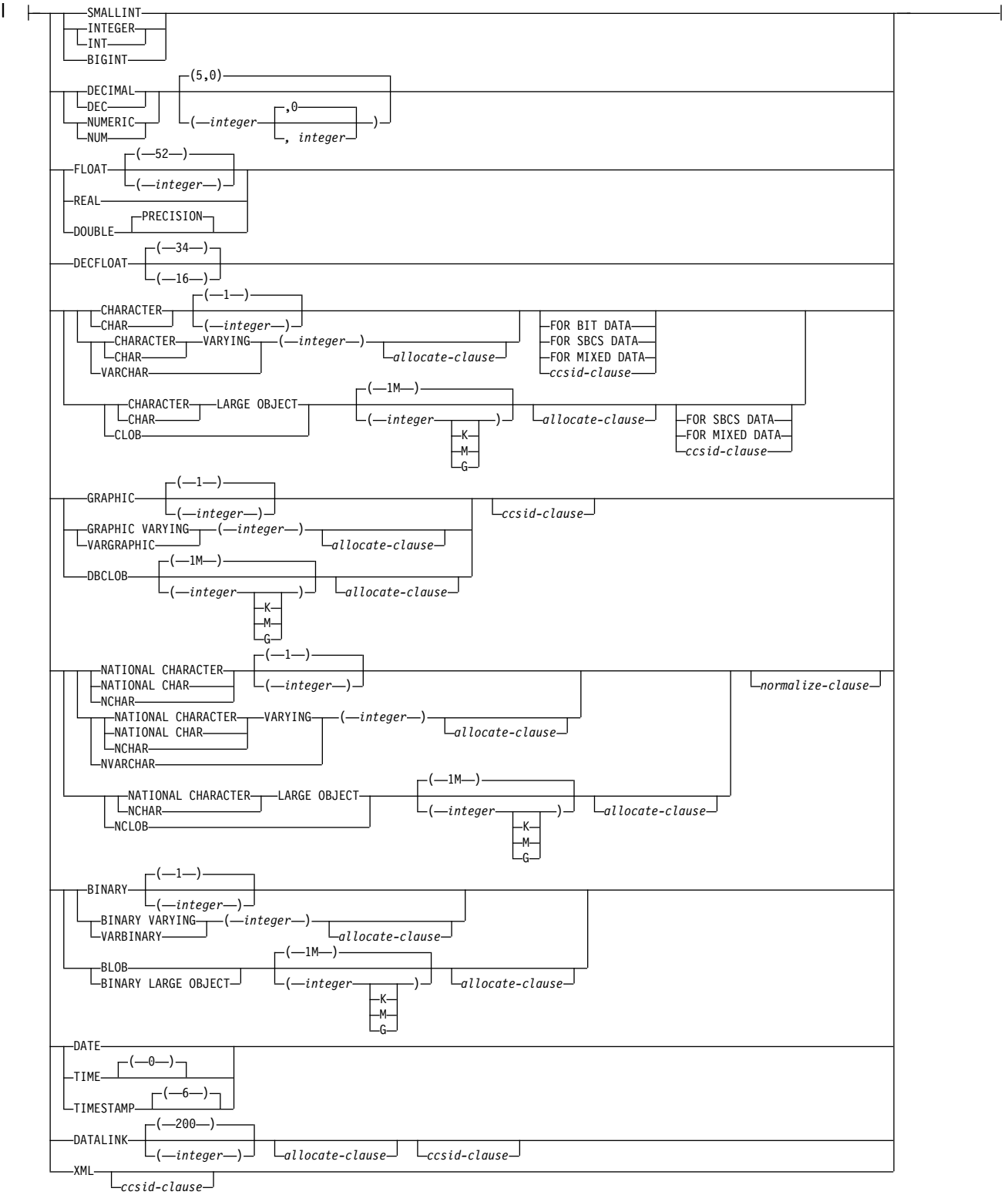
Notes:

1 The same clause must not be specified more than once.

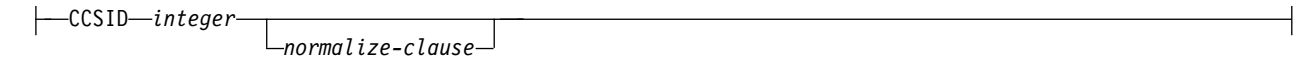
2 *data-type* is optional for row change timestamp columns

3 The datalink-options can only be specified for DATALINKs and distinct-types sourced on DATALINKs.

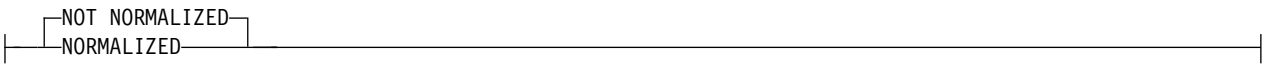
built-in-type:



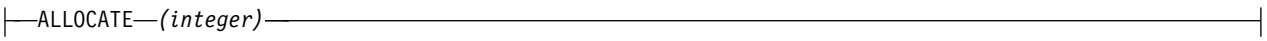
ccsid-clause:



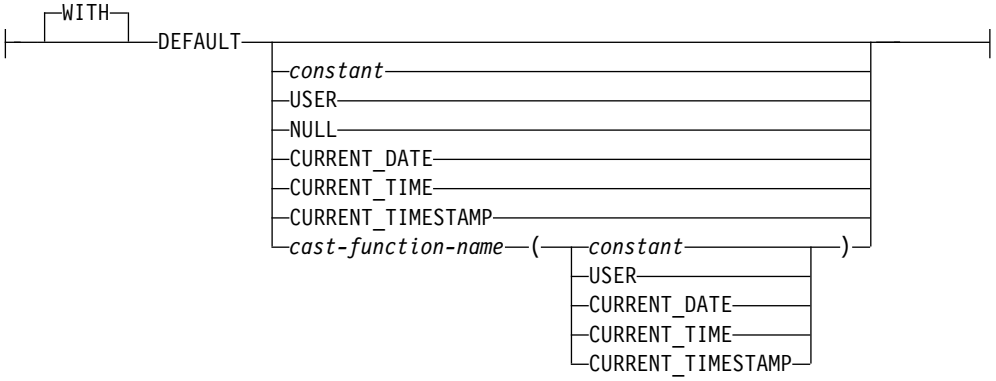
normalize-clause:



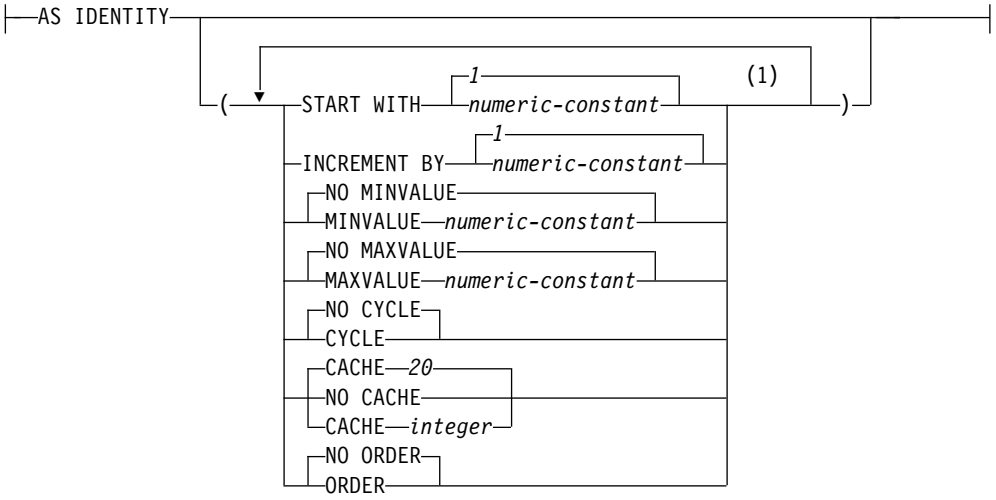
allocate-clause:



default-clause:



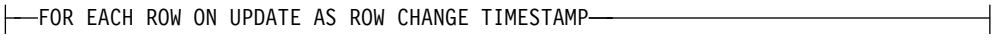
identity-options:



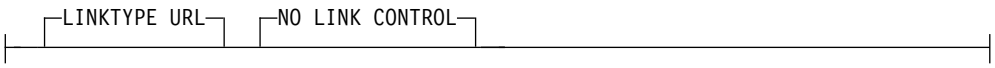
Notes:

1 The same clause must not be specified more than once.

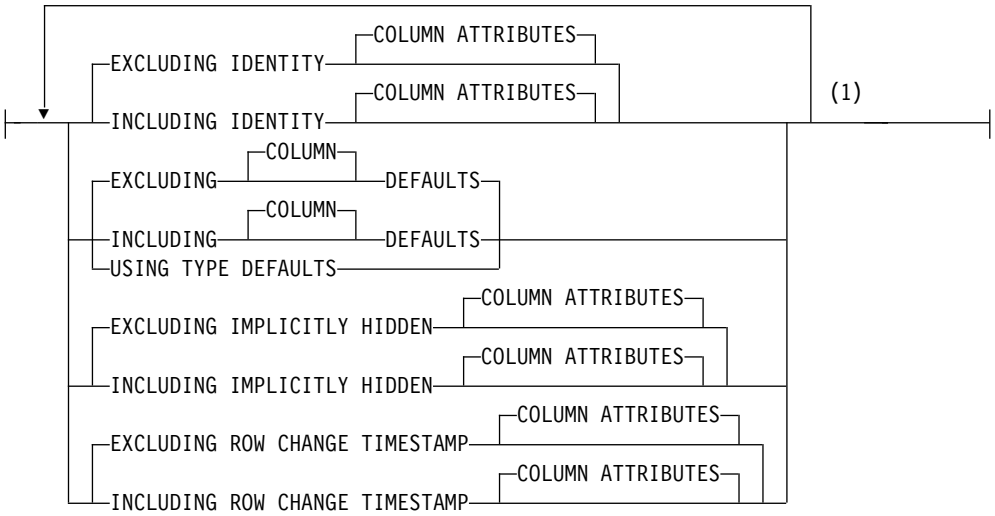
as-row-change-timestamp-clause:



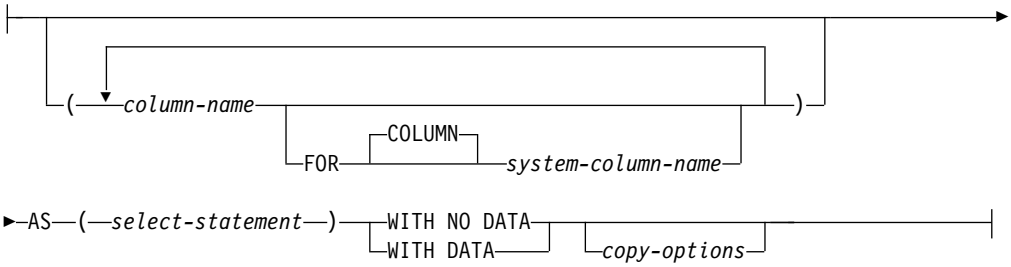
datalink-options:



copy-options:



as-result-table:



Notes:

1 The clauses can be specified in any order.

Description

table-name

Names the temporary table. The qualifier, if specified explicitly, must be SESSION, otherwise an error is returned. If the qualifier is not specified, it is implicitly defined to be SESSION. If a declared temporary table, or an index or view that is dependent on a declared temporary table already exists with the same name, an error is returned.

If a persistent table, view, index, or alias already exists with the same name and the schema name SESSION:

- The declared temporary table is still defined with SESSION.*table-name*. An error is not issued because the resolution of a declared temporary table name does not include a permanent library.

- Any references to SESSION.*table-name* will resolve to the declared temporary table rather than to a permanent table, view, index, or alias with a name of SESSION.*table-name*.

The table will be created in library QTEMP.

FOR SYSTEM NAME *system-object-identifier*
Identifies the *system-object-identifier* of the table. *system-object-identifier* must not be the same as a table, view, alias, or index that already exists at the current server. The *system-object-identifier* must be an unqualified system identifier.

When *system-object-identifier* is specified, *table-name* must not be a valid system object name.

column-definition

Defines the attributes of a column. There must be at least one column definition and no more than 8000 column definitions.

The sum of the row buffer byte counts of the columns must not be greater than 32766 or, if a VARCHAR, VARGRAPHIC, or VARBINARY column is specified, 32740. Additionally, if a LOB or XML column is specified, the sum of the row data byte counts of the columns must not be greater than 3.5 gigabytes. For information about the byte counts of columns according to data type, see "Maximum row sizes" on page 1027.

column-name
Names a column of the table. Do not qualify *column-name* and do not use the same name for more than one column of the table or for a system-column-name of the table.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see "Rules for Column Name Generation" on page 1029.

data-type
Specifies the data type of the column.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 982 for a description of *built-in-type*.

A ROWID column or a DATALINK column with FILE LINK CONTROL cannot be specified for a declared temporary table.

distinct-type-name
Specifies that the data type of the column is a distinct type (a user-defined data type). The length, precision, and scale of the column are respectively the length, precision, and scale of the source type of the distinct type. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path.

DEFAULT
Specifies a default value for the column. This clause cannot be specified more than once in a *column-definition*. DEFAULT cannot be specified for an XML column, an identity column (a column that is defined AS IDENTITY) or a row

change timestamp column. The database manager generates default values for identity columns and row change timestamp columns. For an XML column, the default is NULL unless NOT NULL is specified; in that case there is no default. If a value is not specified following the DEFAULT keyword, then:

- if the column is nullable, the default value is the null value.
- if the column is not nullable, the default depends on the data type of the column:

Data type	Default value
Numeric	0
Fixed-length character or graphic string	Blanks
Fixed-length binary string	Hexadecimal zeros
Varying-length string	A string length of 0
Date	The current date at the time of INSERT
Time	The current time at the time of INSERT
Timestamp	The current timestamp at the time of INSERT
Datalink	A value corresponding to DLVALUE('','URL','')
distinct-type	The default value of the corresponding source type of the distinct type.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

constant
Specifies the constant as the default for the column. The specified constant must represent a value that could be assigned to the column in accordance with the rules of assignment as described in "Assignments and comparisons" on page 98. A floating-point constant or decimal floating-point constant must not be used for a SMALLINT, INTEGER, DECIMAL, or NUMERIC column. A decimal constant must not contain more digits to the right of the decimal point than the specified scale of the column.

USER
Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value of the column. The data type of the column must be CHAR or VARCHAR with a length attribute greater than or equal to the length attribute of the USER special register.

NULL
Specifies null as the default for the column. If NOT NULL is specified, DEFAULT NULL must not be specified within the same *column-definition*.

CURRENT_DATE
Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the column must be DATE or a distinct type based on a DATE.

CURRENT_TIME
Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the column must be TIME or a distinct type based on a TIME.

CURRENT_TIMESTAMP
Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the column must be TIMESTAMP or a distinct type based on a TIMESTAMP.

cast-function-name
This form of a default value can only be used with columns defined as a distinct type, BINARY, VARBINARY, BLOB, CLOB, DBCLOB, DATE, TIME or TIMESTAMP data types. The following table describes the allowed uses of these *cast-functions*.

Data Type	Cast Function Name
Distinct type N based on a BINARY, VARBINARY, BLOB, CLOB, or DBCLOB	BINARY, VARBINARY, BLOB, CLOB, or DBCLOB *
Distinct type N based on a DATE, TIME, or TIMESTAMP	N (the user-defined cast function that was generated when N was created) ** or DATE, TIME, or TIMESTAMP *
Distinct type N based on other data types	N (the user-defined cast function that was generated when N was created) **
BINARY, VARBINARY, BLOB, CLOB, or DBCLOB	BINARY, VARBINARY, BLOB, CLOB, or DBCLOB *
DATE, TIME, or TIMESTAMP	DATE, TIME, or TIMESTAMP *
Notes: * The name of the function must match the name of the data type (or the source type of the distinct type) with an implicit or explicit schema name of QSYS2. ** The name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type.	

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. For BINARY, VARBINARY, BLOB, CLOB, DBCLOB, DATE, TIME, and TIMESTAMP functions, the constant must be a string constant.

USER
Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value for the column. The data type of the source type of the distinct type of the column must be CHAR or VARCHAR with a length attribute greater than or equal to the length attribute of the USER special register.

CURRENT_DATE
Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the source type of the distinct type of the column must be DATE.

CURRENT_TIME
Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the source type of the distinct type of the column must be TIME.

CURRENT_TIMESTAMP
Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the source type of the distinct type of the column must be TIMESTAMP.

If the value specified is not valid, an error is returned.

GENERATED
Specifies that the database manager generates values for the column. GENERATED may be specified if the column is to be considered an identity column or a row change timestamp column.

ALWAYS
Specifies that the database manager will always generate a value for the column when a row is inserted or updated and a default value must be generated. ALWAYS is the recommended value.

BY DEFAULT
Specifies that the database manager will generate a value for the column when a row is inserted or updated and a default value must be generated, unless an explicit value is specified.

For an identity column or row change timestamp column, the database manager inserts or updates a specified value but does not verify that it is a unique value for the column unless the identity column or row change timestamp column has a unique constraint or a unique index that solely specifies the identity column or row change timestamp column.

AS IDENTITY
Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL or NUMERIC with a scale of zero, or a distinct type based on one of these data types). If a DECIMAL or NUMERIC data type is specified, the precision must not be greater than 31.

An identity column is implicitly NOT NULL. An identity column cannot have a DEFAULT clause. See the AS IDENTITY clause in "CREATE TABLE" on page 982 for the descriptions of the identity attributes.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP
Specifies that the column is a timestamp and the values will be generated by the database manager. The database manager generates a value for the column for each row as a row is inserted, and for every row in which any column is updated. The value generated for a row change timestamp column is a timestamp corresponding to the time of the insert or update of the row. If multiple rows are inserted with a single SQL statement, the value for the row change timestamp column may be different for each row to reflect when each row was inserted. The generated value is not guaranteed to be unique.

A table can have only one row change timestamp column. If *data-type* is specified, it must be a TIMESTAMP or a distinct type based on a TIMESTAMP. A row change timestamp column cannot have a DEFAULT clause and must be NOT NULL.

FIELDPROC
Designates an *external-program-name* as the field procedure exit routine for the column. It must be an ILE program that does not contain SQL. It cannot be a service program.

The field procedure encodes and decodes column values. Before a value is inserted in the column, it is passed to the field procedure for encoding. Before a value from the column is used, it is passed to the field procedure for decoding.

The field procedure is also invoked during the processing of the CREATE TABLE statement. When so invoked, the procedure provides DB2 with the column's field description. The field description defines the data characteristics of the encoded values. By contrast, the information supplied for the column in the CREATE TABLE statement defines the data characteristics of the decoded values.

constant
Specifies a parameter that is passed to the field procedure when it is invoked. A parameter list is optional.

A field procedure cannot be defined for a column that is a ROWID or DATALINK or a distinct type based on a ROWID or DATALINK. The column must not be an identity column or a row change timestamp column. The column must not have a default value of CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP, or USER. The column cannot be referenced in a check condition. If it is part of a foreign key, the corresponding parent key column must use the same field procedure. See Embedded SQL programming topic collection for an example of a field procedure.

NOT NULL
Prevents the column from containing null values. Omission of NOT NULL implies that the column can be null. NOT NULL is required for a row change timestamp column.

NOT HIDDEN
Indicates the column is included in implicit references to the table in SQL statements. This is the default.

IMPLICITLY HIDDEN
Indicates the column is not visible in SQL statements unless it is referred to explicitly by name. For example, SELECT * does not include any hidden columns in the result. A table must contain at least one column that is not IMPLICITLY HIDDEN.

datalink-options
Specifies the options associated with a DATALINK data type.

LINKTYPE URL
Defines the type of link as a Uniform Resource Locator (URL).

NO LINK CONTROL
Specifies that there will not be any check made to determine that the linked files exist. Only the syntax of the URL will be checked. There is no database manager control over the linked files.

LIKE

table-name **or** *view-name*
Specifies that the columns defined in the specified table or view are included in this table. The *table-name* or *view-name* specified in a LIKE clause must identify the table or view that already exists at the application server.

The use of LIKE is an implicit definition of n columns, where n is the number of columns in the identified table or view. The implicit definition includes the following attributes of the n columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID

If the LIKE clause is specified immediately following the *table-name* and not enclosed in parenthesis, the following column attributes are also included, otherwise they are not included (the default value, identity, row change timestamp, and hidden attributes can also be controlled by using the *copy-options*):

- Default value, if a *table-name* is specified (*view-name* is not specified)
- Identity attributes
- Nullability
- Hidden attributes
- Row change timestamp attribute
- Column heading and text (see "LABEL" on page 1263)

If the specified table or view is a non-SQL created physical file or logical file, any non-SQL attributes are removed. For example, the date and time format will be changed to ISO.

The implicit definition does not include any other optional attributes of the identified table or view. For example, the new table does not automatically include a primary key or foreign key from a table. The new table has these and other optional attributes only if the optional clauses are explicitly specified.

as-result-table

column-name
Names a column of the table. Do not qualify *column-name* and do not use the same name for more than one column of the table or for a system-column-name of the table.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see "Rules for Column Name Generation" on page 1029.

select-statement
Specifies that the columns of the table are to have the same name and description as the columns that would appear in the derived result table of the *select-statement* if the *select-statement* were to be executed. The use of AS *select-statement* is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *select-statement*. The implicit definition includes the following attributes of the *n* columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID
- Nullability
- Column heading and text (see "LABEL" on page 1263)

The following attributes are not included (the default value, identity, row change timestamp, and hidden attributes may be included by using the *copy-options*):

- Default value
- Hidden attribute
- Identity attributes
- Row change timestamp attribute

The implicit definition does not include any other optional attributes of the tables or views referenced in the *select-statement*.

The implicitly defined columns of the table inherit the names of the columns from the result table of the *select-statement*. Therefore, a column name must be specified in the *select-statement* or in the column name list for all result columns. For result columns that are derived from expressions, constants, and functions, the *select-statement* must include the AS column-name clause immediately after the result column or a name must be specified in the column list preceding the *select-statement*.

The *select-statement* must not refer to variables or include parameter markers (question marks). The *select-statement* must not contain a PREVIOUS VALUE or a NEXT VALUE expression. The UPDATE, SKIP LOCKED DATA, and USE AND KEEP EXCLUSIVE LOCKS clauses may not be specified.

If the *select-statement* contains an *isolation-clause* the isolation level specified in the *isolation-clause* applies to the entire SQL statement.

WITH DATA
Specifies that the *select-statement* is executed. After the table is created, the result table rows of the *select-statement* are automatically inserted into the table.

WITH NO DATA
Specifies that the *select-statement* is not executed. Therefore, there is no result table with a set of rows with which to automatically populate the table.

copy-options

INCLUDING IDENTITY COLUMN ATTRIBUTES or EXCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies whether identity column attributes are inherited.

INCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies that the table inherits the identity attribute, if any, of the columns resulting from *select-statement*, *table-name*, or *view-name*. In general, the identity attribute is copied if the element of the corresponding column in the table, view, or *select-statement* is the name of a table column or the name of a view column that directly or indirectly maps to the name of a base table column with the identity attribute. If the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified with the AS *select-statement* clause, the columns of the new table do not inherit the identity attribute in the following cases:

- The select list of the *select-statement* includes multiple instances of an identity column name (that is, selecting the same column more than once).
- The select list of the *select-statement* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *select-statement* includes a set operation (UNION or INTERSECT).

If INCLUDING IDENTITY is not specified, the table will not have an identity column.

EXCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies that the table does not inherit the identity attribute, if any, of the columns resulting from the *fullselect*, *table-name*, or *view-name*.

EXCLUDING COLUMN DEFAULTS or INCLUDING COLUMN DEFAULTS or USING TYPE DEFAULTS
Specifies whether column defaults are inherited.

EXCLUDING COLUMN DEFAULTS
Specifies that the column defaults are not inherited from the definition of the source table. The default values of the column of the new table are either null or there are no default values. If the column can be null, the default is the null value. If the column cannot be null, there is no default value, and an error occurs if a value is not provided for a column on INSERT for the new table.

INCLUDING COLUMN DEFAULTS
Specifies that the table inherits the default values of the columns resulting from the *select-statement*, *table-name*, or *view-name*. A default value is the value assigned to a column when a value is not specified on an INSERT.

Do not specify INCLUDING COLUMN DEFAULTS, if you specify USING TYPE DEFAULTS.

If INCLUDING COLUMN DEFAULTS is not specified, the default values are not inherited.

USING TYPE DEFAULTS
Specifies that the default values for the table depend on the data type of the columns that result from the *select-statement*, *table-name*, or *view-name*. If the column is nullable, then the default value is the null value. Otherwise, the default value is as follows:

Data type	Default value
Numeric	0
Fixed-length character or graphic string	Blanks
Fixed-length binary string	Hexadecimal zeros
Varying-length string	A string length of 0
Date	The current date at the time of INSERT
Time	The current time at the time of INSERT
Timestamp	The current timestamp at the time of INSERT
Datalink	A value corresponding to DLVALUE('','URL','')
distinct-type	The default value of the corresponding source type of the distinct type.

Do not specify USING TYPE DEFAULTS if INCLUDING COLUMN DEFAULTS is specified.

INCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES or EXCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES
Specifies whether implicitly hidden columns are inherited.

INCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES
Specifies that the table inherits implicitly hidden columns from *select-statement*, *table-name*, or *view-name* and those columns will be defined with the implicitly hidden attribute in the new table.

If INCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES is not specified, the table will not have any implicitly hidden columns.

EXCLUDING IMPLICITLY HIDDEN COLUMN ATTRIBUTES
Specifies that the table does not inherit implicitly hidden columns from the *fullselect*, *table-name*, or *view-name*.

INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES or EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies whether the row change timestamp attribute is inherited.

INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies that the table inherits the row change timestamp attribute, if any, of the columns resulting from *select-statement*, *table-name*, or *view-name*. In general, the row change timestamp attribute is copied if the element of the corresponding column in the table, view, or *select-statement* is the name of a table column or the name of a view column that directly or indirectly maps to the name of a base table column with the row change timestamp attribute. If the INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES clause is specified with the AS *select-statement* clause, the columns of the new table do not inherit the row change timestamp in the following cases:

- The select list of the *select-statement* includes multiple instances of a row change timestamp column name (that is, selecting the same column more than once).
- The select list of the *select-statement* includes multiple row change timestamp columns (that is, it involves a join).
- The row change timestamp column is included in an expression in the select list.
- The *select-statement* includes a set operation (UNION or INTERSECT).

If INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES is not specified, the table will not have a row change timestamp column.

EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies that the table does not inherit the row change timestamp attribute, if any, of the columns resulting from the *fullselect*, *table-name*, or *view-name*.

WITH REPLACE
Specifies that, in the case that a declared temporary table already exists with the specified name, the existing table is replaced with the temporary table defined by this statement (and all rows of the existing table are deleted).

When WITH REPLACE is not specified, then the name specified must not identify a declared temporary table that already exists in the current session.

ON COMMIT
Specifies the action taken on the declared temporary table when a COMMIT operation is performed. The default is DELETE ROWS.

The ON COMMIT clause does not apply if the declared temporary table is opened under isolation level No Commit (NC) or if a COMMIT HOLD operation is performed.

DELETE ROWS
All rows of the table will be deleted if no WITH HOLD cursor is open on the table.

PRESERVE ROWS
Rows of the table will be preserved.

NOT LOGGED
Changes to the table are not logged, including creation of the table. When a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed and the table was changed in the unit of work (or savepoint), the changes are not rolled back. If the table was created in the unit of work (or savepoint), then that table will be dropped. If the table was dropped in the unit of work (or savepoint) then the table will be restored, but with no rows.

ON ROLLBACK
Specifies the action taken on the declared temporary table when a ROLLBACK operation is performed.

The ON ROLLBACK clause does not apply if the declared temporary table was opened under isolation level No Commit (NC) or if a ROLLBACK HOLD operation is performed.

DELETE ROWS
All rows of the table will be deleted. This is the default.

PRESERVE ROWS
Rows of the table will be preserved.

RCDFMT *format-name*
An unqualified name that designates the IBM i record format name of the table. A *format-name* is a system identifier.

If a record format name is not specified, the *format-name* is the same as the *system-object-name* of the table.

Notes

Instantiation, scope, and termination: Let P denote an application process and let T be a declared temporary table in an application program in P:

- When a program in P issues a DECLARE GLOBAL TEMPORARY TABLE statement, an empty instance of T is created.
- Any program in P can reference T, and any of those references is a reference to that same instance of T. (If a DECLARE GLOBAL TEMPORARY statement is specified within a compound statement of an SQL function, SQL procedure, or trigger; the scope of the declared temporary table is the application process and not the compound statement.)

  If T was declared at a remote server, the reference to T must use the same connection that was used to declare T and that connection must not have been terminated after T was declared. When the connection to the database server at which T was declared terminates, T is dropped.
- If T is defined with the ON COMMIT DELETE ROWS clause, when a commit operation terminates a unit of work in P and no program in P has a WITH HOLD cursor open that is dependent on T, all rows are deleted.
- If T is defined with the ON ROLLBACK DELETE ROWS clause, when a rollback operation terminates a unit of work in P, all rows are deleted.
- When the application process that declared T terminates, T is dropped.

Temporary table ownership: The *owner* of the table is the user profile of the job executing the statement.

Temporary table authority: When a declared temporary table is defined, PUBLIC implicitly is granted all table privileges on the table and authority to drop the table.

Referring to a declared temporary table in other SQL statements: Many SQL statements support declared temporary tables. To refer to a declared temporary table in an SQL statement other than DECLARE GLOBAL TEMPORARY TABLE, the table must be implicitly or explicitly qualified with SESSION.

If you use SESSION as the qualifier for a table name but the application process does not include a DECLARE GLOBAL TEMPORARY TABLE statement for the table name, the database manager assumes that you are not referring to a declared temporary table. The database manager resolves such table references to a permanent table.

Restrictions on the use of declared temporary tables:

- Declared temporary tables cannot be specified in an ALTER TABLE, COMMENT, CREATE ALIAS, GRANT, LABEL, LOCK, RENAME, or REVOKE statement.
- Declared temporary tables cannot be specified as the parent table in referential constraints
- If a declared temporary table is referenced in a CREATE INDEX or CREATE VIEW statement, the index or view must be created in SESSION (or library QTEMP).

Creating a table using a remote select-statement: The *select-statement* for an *as-result-table* can refer to tables on a different server than where the table is being created. This can be done by using a three-part object name or an alias that is defined to reference a three-part name of a table or view. The result of the *select-statement* cannot contain a column that has a field procedure defined.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- INLINE LENGTH is a synonym for ALLOCATE.
- DEFINITION ONLY is a synonym for WITH NO DATA

Examples

Example 1: Define a declared temporary table with column definitions for an employee number, salary, commission, and bonus.

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP_EMP
  (EMPNO    CHAR(6)     NOT NULL,
   SALARY    DECIMAL(9, 2),
   BONUS     DECIMAL(9, 2),
   COMM      DECIMAL(9, 2))
  ON COMMIT PRESERVE ROWS
```

Example 2: Assume that base table USER1.EMPTAB exists and that it contains three columns, one of which is an identity column. Declare a temporary table that has the same column names and attributes (including identity attributes) as the base table.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPTAB1
  LIKE USER1.EMPTAB
  INCLUDING IDENTITY
  ON COMMIT PRESERVE ROWS
```

In the above example, the database manager uses SESSION as the implicit qualifier for TEMPTAB1.

DECLARE PROCEDURE

The DECLARE PROCEDURE statement defines an external procedure.

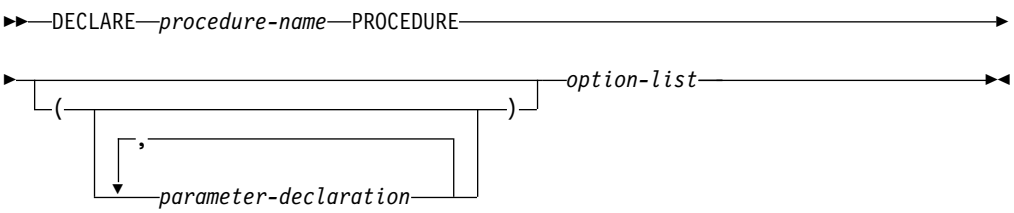
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in REXX.

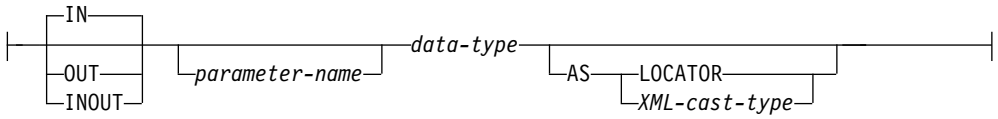
Authorization

None.

Syntax

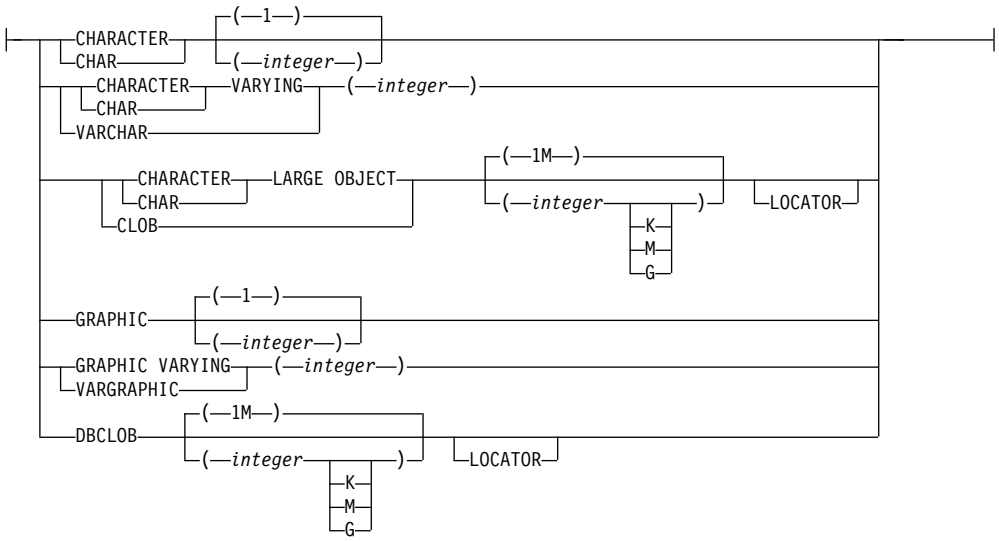


parameter-declaration:

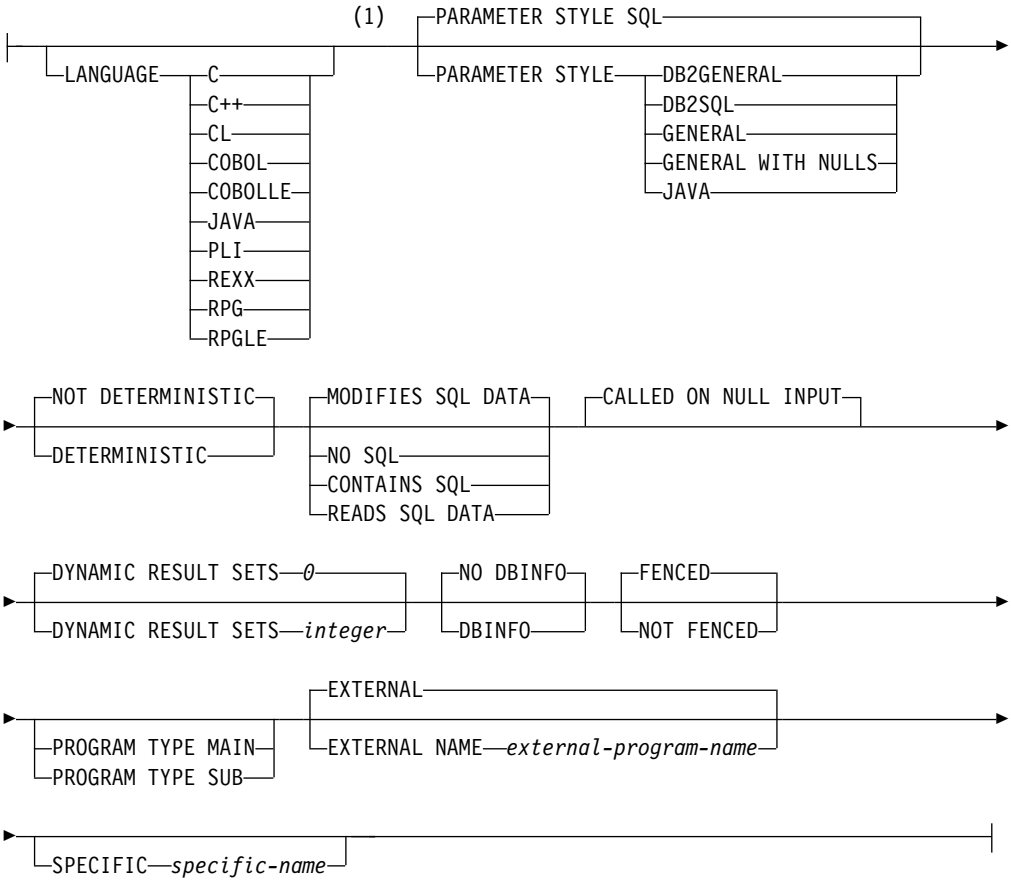


data-type:

XML-cast-type:



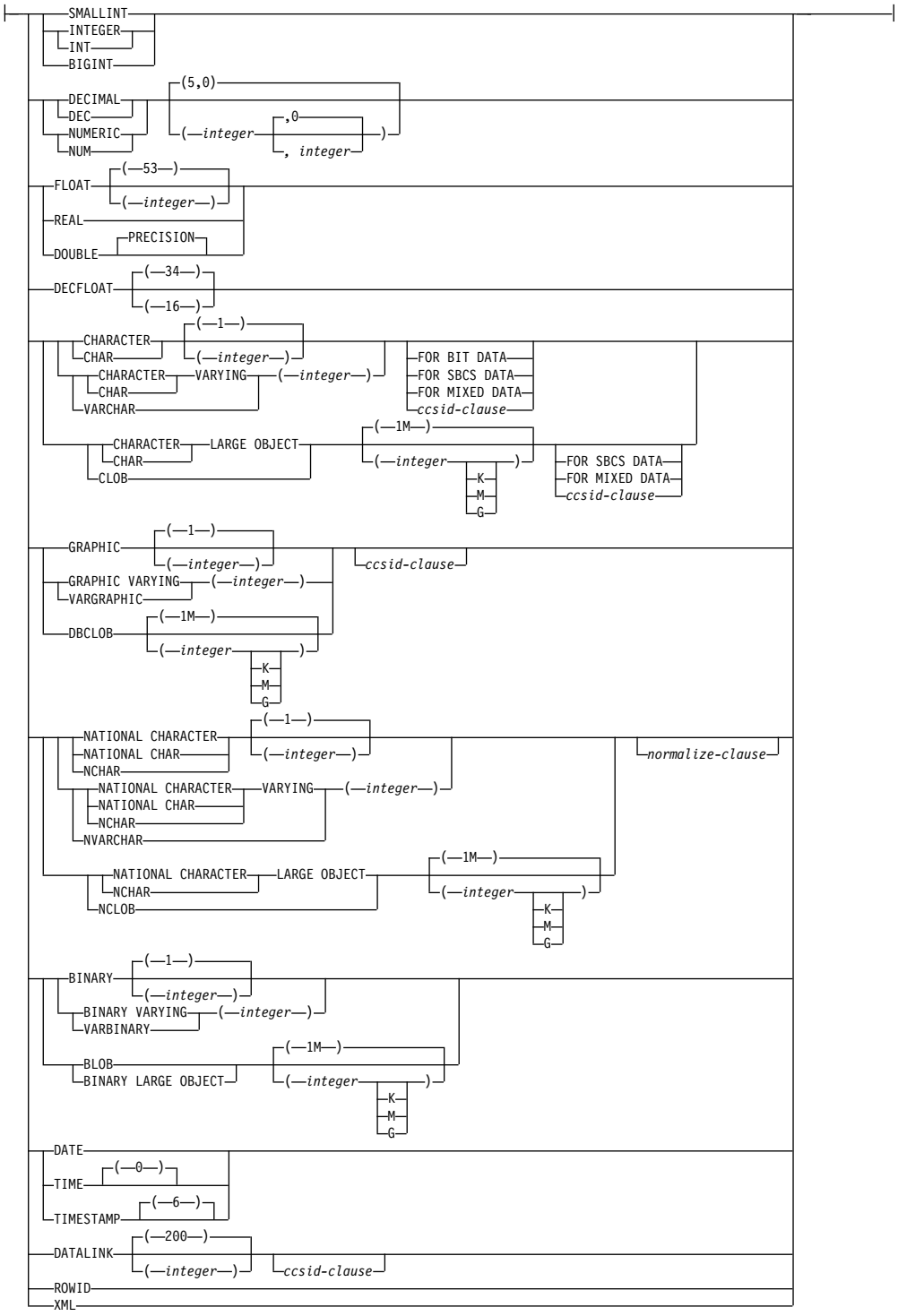
option-list:



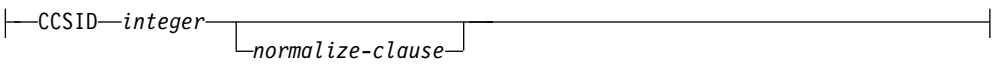
Notes:

1 The optional clauses can be specified in a different order.

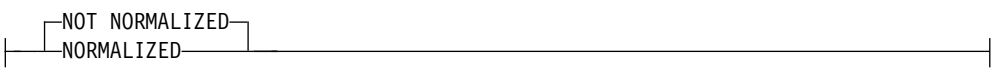
built-in-type:



ccsid-clause:



normalize-clause:



Description

procedure-name
Names the procedure. The name must not be the same as the name of another procedure declared in your source program.

(*parameter-declaration***,...)**
Specifies the number of parameters of the procedure and the data type of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. Although not required, you can give each parameter a name.

The maximum number of parameters allowed in DECLARE PROCEDURE depends on the language and the parameter style:

- If PARAMETER STYLE GENERAL is specified, in C and C++, the maximum is 1024. Otherwise, the maximum is 255.
- If PARAMETER STYLE GENERAL WITH NULLS is specified, in C and C++, the maximum is 1023. Otherwise, the maximum is 254.
- If PARAMETER STYLE SQL or PARAMETER STYLE DB2SQL is specified, in C and C++, the maximum is 508. Otherwise, the maximum is 90.
- If PARAMETER STYLE JAVA or PARAMETER STYLE DB2GENERAL is specified, the maximum is 90.

The maximum number of parameters is also limited by the maximum number of parameters allowed by the licensed program used to compile the external program or service program.

IN
Identifies the parameter as an input parameter to the procedure. Any changes made to the parameter within the procedure are not available to the calling SQL application when control is returned.[96]

OUT
Identifies the parameter as an output parameter that is returned by the procedure.

A DataLink or a distinct type based on a DataLink may not be specified as an output parameter.

INOUT
Identifies the parameter as both an input and output parameter for the procedure.

A DataLink or a distinct type based on a DataLink may not be specified as an input and output parameter.

parameter-name
Names the parameter. The name cannot be the same as any other *parameter-name* for the procedure.

data-type
Specifies the data type of the parameter.

96. When the language type is REXX, all parameters must be input parameters.

The data type must be valid for the language specified in the language clause. All data types are valid for SQL procedures. DataLinks are not valid for external procedures. For more information about data types, see "CREATE TABLE" on page 982, and the SQL Programming topic collection.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the procedure. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the procedure is called.

Any parameter that has an XML type must specify the *XML-cast-type* clause.

AS LOCATOR
Specifies that the parameter is a locator to the value rather than the actual value. You can specify AS LOCATOR only if the parameter has a LOB or XML data type or a distinct type based on a LOB or XML data type. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

AS *XML-cast-type*
Specifies the data type passed to the procedure for a parameter that is XML type or a distinct type based on XML type. If LOCATOR is specified, the parameter is a locator to the value rather than the actual value.

If a CCSID value is specified, only Unicode CCSID values can be specified for graphic data types. If a CCSID value is not specified, the CCSID is established at the time the containing program, module, or service program is created according to the SQL_XML_DATA_CCSID QAQQINI option setting. The default CCSID is 1208. See "XML Values" on page 88 for a description of this option.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of result sets that can be returned from the procedure. *integer* must be greater than or equal to zero and less than 32768. If zero is specified, no result sets are returned. If the SET RESULT SETS statement is issued, the number of results returned is the minimum of the number of result sets specified on this keyword and the SET RESULT SETS statement.

For more information about result sets, see "SET RESULT SETS" on page 1390.

LANGUAGE
Specifies the language that the external program is written in. The language clause is required if the external program is a REXX procedure.

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program. If the program attribute information associated with the program does not identify a recognizable language, then the language is assumed to be C.

C
The external program is written in C.

C++
The external program is written in C++.

CL
The external program is written in CL.

COBOL
The external program is written in COBOL.

COBOLLE
The external program is written in ILE COBOL.

JAVA
The external program is written in JAVA.

PLI
The external program is written in PL/I.

REXX
The external program is a REXX procedure.

RPG
The external program is written in RPG.

RPGLE
The external program is written in ILE RPG.

PARAMETER STYLE
Specifies the conventions used for passing parameters to and returning the values from procedures:

SQL
Specifies that in addition to the parameters on the CALL statement, several additional parameters are passed to the procedure. The parameters are defined to be in the following order:

- The first N parameters are the parameters that are specified on the DECLARE PROCEDURE statement.
- N parameters for indicator variables for the parameters.
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the procedure. The SQLSTATE returned is assigned by the external program.

  The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the function.
- A VARCHAR(517) input parameter for the fully qualified procedure name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(1000) output parameter for the message text.

These parameters are passed according to the specified LANGUAGE. For example, if the language is C or C++, the VARCHAR parameters are passed as NUL-terminated strings. For more information about the parameters passed, see the include sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

PARAMETER STYLE SQL cannot be used with LANGUAGE JAVA.

DB2GENERAL
Specifies that the procedure will use a parameter passing convention that is defined for use with Java methods.

PARAMETER STYLE DB2GENERAL can only be specified with LANGUAGE JAVA. For details on passing parameters in JAVA, see the IBM Developer Kit for Java topic collection.

DB2SQL
Specifies that in addition to the parameters on the CALL statement, several

additional parameters are passed to the procedure. DB2SQL is identical to the SQL parameter style, except that the following additional parameter may be passed as the last parameter:

- A parameter for the dbinfo structure, if DBINFO was specified on the DECLARE PROCEDURE statement.

For more information about the parameters passed, see the include sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

PARAMETER STYLE DB2SQL cannot be used with LANGUAGE JAVA.

GENERAL
Specifies that the procedure will use a parameter passing mechanism where the procedure receives the parameters specified on the CALL. Additional arguments are not passed for indicator variables.

PARAMETER STYLE GENERAL cannot be used with LANGUAGE JAVA.

GENERAL WITH NULLS
Specifies that in addition to the parameters on the CALL statement as specified in GENERAL, another argument is passed to the procedure. This additional argument contains an indicator array with an element for each of the parameters of the CALL statement. In C, this would be an array of short INTs. For more information about how the indicators are handled, see the SQL Programming topic collection.

PARAMETER STYLE GENERAL WITH NULLS cannot be used with LANGUAGE JAVA.

JAVA
Specifies that the procedure will use a parameter passing convention that conforms to the Java language and SQLJ Routines specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values. For increased portability, you should write Java procedures that use the PARAMETER STYLE JAVA conventions.

PARAMETER STYLE JAVA can only be specified with LANGUAGE JAVA. For details on passing parameters in JAVA, see the IBM Developer Kit for Java topic collection.

Note that the language of the external function determines how the parameters are passed. For example, in C, any VARCHAR or CHAR parameters are passed as NUL-terminated strings. For more information, see SQL Programming. For Java routines, see IBM Developer Kit for Java.

SPECIFIC *specific-name*
Specifies a qualified or unqualified name that uniquely identifies the procedure. The *specific-name*, including the implicit or explicit qualifier, must be the same as the *procedure-name*.

If no qualifier is specified, the implicit or explicit qualifier of the *procedure-name* is used. If a qualifier is specified, the qualifier must be the same as the explicit or implicit qualifier of the *procedure-name*.

If *specific-name* is not specified, it is the same as the procedure name.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments.

NOT DETERMINISTIC
The procedure may not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

DETERMINISTIC
The procedure always returns the same results each time the procedure is called with the same IN and INOUT arguments, provided the referenced data in the database has not changed.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL
Specifies which SQL statements, if any, may be executed in the procedure or any routine called from this procedure. The default is MODIFIES SQL DATA. See Appendix B, "Characteristics of SQL statements," on page 1501 for a detailed list of the SQL statements that can be executed under each data access indication.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA
Specifies that SQL statements that do not modify SQL data can be included in the procedure.

CONTAINS SQL
Specifies that SQL statements that neither read nor modify SQL data can be executed by the procedure.

NO SQL
Specifies that the procedure cannot execute any SQL statements.

CALLED ON NULL INPUT
Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

FENCED or NOT FENCED
This parameter is allowed for compatibility with other products and is not used by DB2 for i.

PROGRAM TYPE MAIN or PROGRAM TYPE SUB
This parameter is allowed for compatibility with other products. It indicates whether the routine's external program is a program (*PGM) or a procedure in a service program (*SRVPGM).

PROGRAM TYPE MAIN
Specifies that the routine executes as the main entry point in a program. The external program must be a *PGM object.

PROGRAM TYPE SUB
Specifies that the procedure executes as a procedure in a service program. The external program must be a *SRVPGM object.

DBINFO
Specifies that the database manager should pass a structure containing status information to the procedure. Table 80 on page 1114 contains a description of the DBINFO structure. Detailed information about the DBINFO structure can be found in include sqludf in the appropriate source file in library QSYSINC. For example, for C, sqludf can be found in QSYSINC/H.

DBINFO is only allowed with PARAMETER STYLE DB2SQL.

Table 80. DBINFO fields

Field	Data Type	Description
Relational database	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The run-time authorization ID.
CCSID Information	INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER CHAR(8)	The CCSID information of the job. Three sets of three CCSIDs are returned. The following information identifies the three CCSIDs in each set: • SBCS CCSID • DBCS CCSID • Mixed CCSID Following the three sets of CCSIDs is an integer that indicates which set of three sets of CCSIDs is applicable and eight bytes of reserved space. Each set of CCSIDs is for a different encoding scheme (EBCDIC, ASCII, and Unicode). If a CCSID is not explicitly specified for a parameter on the CREATE PROCEDURE statement, the input string is assumed to be encoded in the CCSID of the job at the time the procedure is executed. If the CCSID of the input string is not the same as the CCSID of the parameter, the input string passed to the external procedure will be converted before calling the external program.
Target Column	VARCHAR(128) VARCHAR(128) VARCHAR(128)	Not applicable for a call to a procedure.
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server's platform type.

EXTERNAL NAME *external-program-name*

Specifies the program that will be executed when the procedure is called by the CALL statement. The program name must identify a program that exists at the application server. The program cannot be an ILE service program.

The validity of the name is checked at the application server. If the format of the name is not correct, an error is returned.

If external-program-name is not specified, the external program name is assumed to be the same as the procedure name.

Notes

DECLARE PROCEDURE scope: The scope of the *procedure-name* is the source program in which it is defined; that is, the program submitted to the precompiler. Thus, a program called from another separately compiled program or module will not use the attributes from a DECLARE PROCEDURE statement in the calling program.

DECLARE PROCEDURE rules: The DECLARE PROCEDURE statement should precede all CALL statements that reference that procedure.

The DECLARE PROCEDURE statement only applies to static CALL statements. It does not apply to any dynamically prepared CALL statements or a CALL statement where the procedure name is identified by a variable.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL can be used as synonyms for CALLED ON NULL INPUT.
- The keywords SIMPLE CALL can be used as a synonym for GENERAL.
- The value DB2GENRL may be used as a synonym for DB2GENERAL.
- The keywords PARAMETER STYLE in the PARAMETER STYLE clause are optional.

Example

Declare an external procedure PROC1 in a C program. When the procedure is called using the CALL statement, a COBOL program named PGM1 in library LIB1 will be called.

```
EXEC SQL
DECLARE PROC1 PROCEDURE
        (CHAR(10), CHAR(10))
        EXTERNAL NAME LIB1.PGM1
        LANGUAGE COBOL GENERAL;

EXEC SQL
CALL PROC1 ('FIRSTNAME ','LASTNAME  ');
```

DECLARE STATEMENT

The DECLARE STATEMENT statement is used for program documentation. It declares names that are used to identify prepared SQL statements.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. This statement is not allowed in Java or REXX.

Authorization

None required.

Syntax

Description

statement-name
Lists one or more names that are used in your program to identify prepared SQL statements.

Example

This example shows the use of the DECLARE STATEMENT statement in a C program.

```
EXEC SQL INCLUDE SQLDA;
void main ()
  {
   EXEC SQL BEGIN DECLARE SECTION ;
   char src_stmt[32000];
   char sqlda[32000]
   EXEC SQL END DECLARE SECTION ;
   EXEC SQL INCLUDE SQLCA ;

   strcpy(src_stmt,"SELECT DEPTNO, DEPTNAME, MGRNO \
     FROM DEPARTMENT \
     WHERE ADMRDEPT = 'A00'");

   EXEC SQL DECLARE OBJ_STMT STATEMENT;

   (Allocate storage from SQLDA)

   EXEC SQL DECLARE C1 CURSOR FOR OBJ_STMT;

   EXEC SQL PREPARE OBJ_STMT FROM :src_stmt;
   EXEC SQL DESCRIBE OBJ_STMT INTO :sqlda;

   (Examine SQLDA)  (Set SQLDATA pointer addresses)

   EXEC SQL OPEN C1;

   while (strncmp(SQLSTATE, "00000", 5) )
     {
     EXEC SQL FETCH C1 USING DESCRIPTOR :sqlda;
```

```
  (Print results)

  }

EXEC SQL CLOSE C1;
return;
}
```

DECLARE VARIABLE

The DECLARE VARIABLE statement is used to assign a subtype or CCSID other than the default to a host variable.

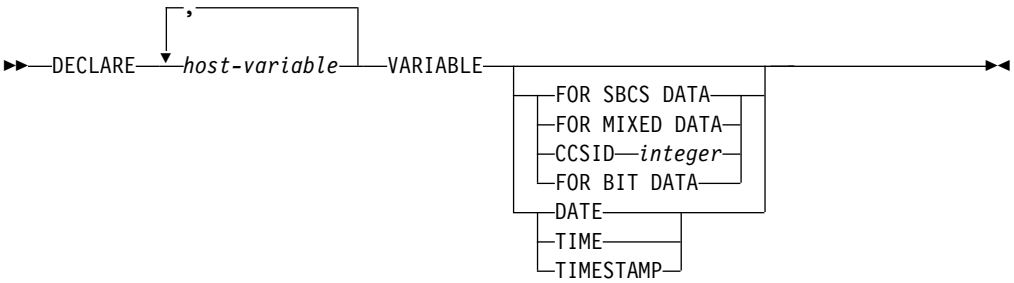
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

host-variable
Names a character, graphic, or XML string host variable defined in the program. An indicator variable cannot be specified for the host-variable. The host-variable definition may either precede or follow a DECLARE VARIABLE statement that refers to that variable.

FOR BIT DATA
Specifies that the values of the host-variable are not associated with a coded character set and, therefore, are never converted. The CCSID of a FOR BIT DATA host variable is 65535. FOR BIT DATA cannot be specified for graphic or XML host-variables.

FOR SBCS DATA
Specifies that the values of the host variable contain SBCS (single-byte character set) data. FOR SBCS DATA is the default if the CCSID attribute of the job at the application requester is not DBCS-capable or if the length of the host variable is less than 4. The CCSID of FOR SBCS DATA is determined by the CCSID attribute of the job at the application requester. FOR SBCS DATA cannot be specified for graphic or XML host-variables.

FOR MIXED DATA
Specifies that the values of the host variable contain both SBCS data and DBCS data. FOR MIXED DATA is the default if the CCSID attribute of the job at the application requester is DBCS-capable and the length of the host variable is greater than 3. The CCSID of FOR DBCS DATA is determined by the CCSID attribute of the job at the application requester. FOR MIXED DATA cannot be specified for graphic or XML host-variables.

CCSID *integer*
Specifies that the values of the host variable contain data of CCSID integer. If the integer is an SBCS CCSID, the host variable is SBCS data. If the integer is a mixed data CCSID, the host variable is mixed data. For character host variables, the CCSID specified must be an SBCS or mixed CCSID.

If the variable has a graphic string data type, the CCSID specified must be a DBCS, UTF-16, or UCS-2 CCSID. For a list of valid CCSIDs, see Appendix E, "CCSID values," on page 1541. Consider specifying CCSID 1200 or 13488 to indicate UTF-16 or UCS-2 data. If a CCSID is not specified, the CCSID of the graphic string variable will be the associated DBCS CCSID for the job.

If the variable has an XML data type, the CCSID specified must be an SBCS, mixed, or Unicode CCSID. The CCSID must be compatible with the XML AS datatype. If a CCSID is not specified, the CCSID value as specified by the SQL_XML_DATA_CCSID QAQQINI setting is used. See "XML Values" on page 88 for more information.

For file reference variables, the CCSID specifies the CCSID of the path and file name, not the data within the file.

DATE
Specifies that the values of the host variable contain data that is a date.

TIME
Specifies that the values of the host variable contain data that is a time.

TIMESTAMP
Specifies that the values of the host variable contain data that is a timestamp.

Notes

Placement restrictions: The DECLARE VARIABLE statement can be specified anywhere in an application program that SQL statements are valid with the following exceptions:

- If the host language is COBOL or RPG, the DECLARE VARIABLE statement must occur before an SQL statement that refers to a host variable specified in the DECLARE VARIABLE statement.
- If DATE, TIME, or TIMESTAMP is specified for a NUL-terminated character string in C, the length of the C declaration will be reduced by one.

Precompiler rules: The following situations result in an error message during precompile:

- A reference is made to a variable that does not exist.
- A reference is made to a numeric variable.
- A reference is made to a variable that has been referred to already.
- A reference is made to a variable that is not unique.
- A reference is made to an ILE RPG variable that is defined as UCS-2.
- A reference is made to an ILE COBOL variable that is defined as NATIONAL.
- The DECLARE VARIABLE statement occurs after an SQL statement where the SQL statement and the DECLARE VARIABLE statement refer to the same variable.
- The FOR BIT DATA, FOR SBCS DATA, or FOR MIXED DATA clause is specified for a graphic or XML host variable.
- A SBCS or mixed CCSID is specified for a graphic host variable.
- A DBCS, UTF-16, or UCS-2 CCSID is specified for a character host variable.

- A DBCS CCSID is specified for an XML host variable.
- DATE, TIME, or TIMESTAMP is specified for a host variable that is not character.
- The length of a host variable used for DATE, TIME, or TIMESTAMP is not long enough for the minimum date, time, or timestamp value.

Example

In this example, declare C program variables *fred* and *pete* as mixed data, and *jean* and *dave* as SBCS data with CCSID 37.

```
void main ()
  {
     EXEC SQL BEGIN DECLARE SECTION;
     char fred[10];
     EXEC SQL DECLARE :fred VARIABLE  FOR MIXED DATA;

     decimal(6,0) mary;
     char pete[4];
     EXEC SQL DECLARE :pete VARIABLE FOR MIXED DATA;

     char jean[30];
     char dave[9];
     EXEC SQL DECLARE :jean, :dave VARIABLE  CCSID 37;
     EXEC SQL END DECLARE SECTION;
     EXEC SQL INCLUDE SQLCA;
   ...

  }
```

DELETE

The DELETE statement deletes rows from a table or view. Deleting a row from a view deletes the row from the table on which the view is based if no INSTEAD OF DELETE trigger is defined for this view. If such a trigger is defined, the trigger will be activated instead.

There are two forms of this statement:

- The *Searched* DELETE form is used to delete one or more rows (optionally determined by a search condition).
- The *Positioned* DELETE form is used to delete exactly one row (as determined by the current position of a cursor).

Invocation

A Searched DELETE statement can be embedded in an application program or issued interactively. A Positioned DELETE must be embedded in an application program. Both forms are executable statements that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table or view identified in the statement:
  - The DELETE privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

If the *search-condition* in a Searched DELETE contains a reference to a column of the table or view, then the privileges held by the authorization ID of the statement must also include one of the following:

- The SELECT privilege on the table or view
- Administrative authority

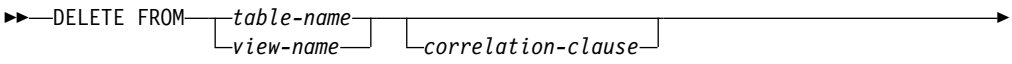
If *search-condition* includes a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following:

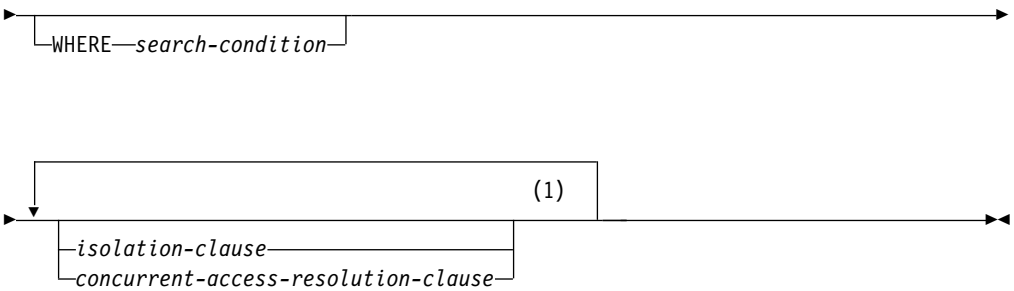
- For each table or view identified in the subquery:
  - The SELECT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

Syntax

Searched DELETE:

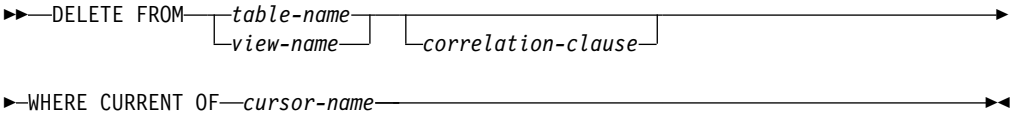




Notes:

1 The same clause must not be specified more than once.

Positioned DELETE:



isolation–clause:

Description

FROM *table-name* **or** *view-name*
Identifies the table or view from which rows are to be deleted. The name must identify a table or view that exists at the current server, but it must not identify a catalog table, a view of a catalog table, or a view that is not deletable. For an explanation of deletable views, see "CREATE VIEW" on page 1069.

correlation-clause
Specifies an alternate name that can be used within the *search-condition* to designate the table or view. For an explanation of *correlation-clause*, see Chapter 5, "Queries," on page 627. For an explanation of *correlation-name*, see "Correlation names" on page 140.

WHERE
Specifies the rows to be deleted. The clause can be omitted, or a *search-condition* or *cursor-name* can be specified. If the clause is omitted, all rows of the table or view are deleted.

search-condition
Is any search condition as described in "Search conditions" on page 225. Each *column-name* in the *search-condition*, other than in a subquery, must identify a column of the table or view.

The *search-condition* is applied to each row of the table or view and the deleted rows are those for which the result of the *search-condition* is true.

If the *search-condition* contains a subquery, the subquery can be thought of as being executed each time the *search condition* is applied to a row, and the results used in applying the *search condition*. In actuality, a subquery with no correlated references may be executed only once, whereas a subquery with a correlated reference may have to be executed once for each row.

If a subquery refers to the object table of the DELETE statement or a dependent table with a delete rule of CASCADE, SET NULL, or SET DEFAULT, the subquery is completely evaluated before any rows are deleted.

CURRENT OF *cursor-name*
Identifies the cursor to be used in the delete operation. The *cursor-name* must identify a declared cursor as explained in the Notes for the DECLARE CURSOR statement.

The table or view identified must also be specified in the FROM clause of the *select-statement* of the cursor and the cursor must be deletable. For an explanation of deletable cursors, see "DECLARE CURSOR" on page 1079.

When the DELETE statement is executed, the cursor must be positioned on a row; that row is the one deleted. After the deletion, the cursor is positioned before the next row of its result table. If there is no next row, the cursor is positioned after the last row.

isolation-clause
Specifies the isolation level to be used for this statement.

WITH

Introduces the isolation level, which may be one of:
- *RR* Repeatable read
- *RS* Read stability
- *CS* Cursor stability
- *UR* Uncommitted read
- *NC* No commit

If *isolation-clause* is not specified the default isolation is used. For more information on the default isolation, see "isolation-clause" on page 693.

concurrent-access-resolution-clause
Specifies the concurrent access resolution to use for the select statement. For more information, see "concurrent-access-resolution-clause" on page 695.

DELETE Rules

Triggers: If the identified table or the base table of the identified view has a delete trigger, the trigger is activated. A trigger might cause other statements to be executed or return error conditions based on the deleted values.

Referential Integrity: If the identified table or the base table of the identified table is a parent table, the rows selected must not have any dependents in a relationship with a delete rule of RESTRICT or NO ACTION, and the DELETE must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT or NO ACTION.

If the delete operation is not prevented by a RESTRICT or NO ACTION delete rule, the selected rows are deleted. Any rows that are dependents of the selected rows are also affected:

- The nullable columns of the foreign keys of any rows that are their dependents in a relationship with a delete rule of SET NULL are set to the null value.
- Any rows that are their dependents in a relationship with a delete rule of CASCADE are also deleted, and the above rules apply, in turn to those rows.
- The columns of the foreign keys of any rows that are their dependents in a relationship with a delete rule of SET DEFAULT are set to the corresponding default value.

The referential constraints (other than a referential constraint with a RESTRICT delete rule), are effectively checked at the end of the statement. In the case of a multiple-row delete, this would occur after all rows were deleted and any associated triggers were activated.

Check Constraints: A check constraint can prevent the deletion of a row in a parent table when there are dependents in a relationship with a delete rule of SET NULL or SET DEFAULT. If deleting a row in the parent table would cause a column in a dependent table to be set to null or a default value and the null or default value would cause a search condition of a check constraint to evaluate to false, the row is not deleted.

Notes

Delete operation errors: If an error occurs while executing any delete operation, changes from this statement, referential constraints, and any triggered SQL statements are rolled back (unless the isolation level is NC for this statement or any other triggered SQL statements).

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful DELETE statement. Until the locks are released by a commit or rollback operation, the effect of the DELETE operation can only be perceived by:

- The application process that performed the deletion
- Another application process using isolation level UR or NC

The locks can prevent other application processes from performing operations on the table. For further information about locking, see the description of the COMMIT, ROLLBACK, and LOCK TABLE statements, and “Isolation level” on page 26.

If an application process deletes a row on which any of its non-updatable cursors are positioned, those cursors are positioned before the next row of their result table. Let C be a cursor that is positioned before the next row R (as the result of an OPEN, a DELETE through C, a DELETE through some other cursor, or a Searched DELETE). In the presence of INSERT, UPDATE, and DELETE operations that affect the base table from which R is derived, the next FETCH operation referencing C does not necessarily position C on R. For example, the operation can position C on R' where R' is a new row that is now the next row of the result table.

A maximum of 4000000 rows can be deleted or changed in any single DELETE statement when COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) was specified. The number of rows changed includes any rows inserted, updated, or deleted under the same commitment definition as a result of a trigger, a CASCADE, SET NULL, or SET DEFAULT referential integrity delete rule.

Position of cursor: If an application process deletes a row on which any of its cursors are positioned, those cursors are positioned before the next row of their result table. Let C be a cursor that is positioned before row R (as a result of an OPEN, a DELETE through C, a DELETE through some other cursor, or a Searched DELETE). In the presence of INSERT, UPDATE, and DELETE operations that affect the base table from which R is derived, the next FETCH operation referencing C does not necessarily position C on R. For example, the operation can position C on R', where R' is a new row that is now the next row of the result table.

Number of rows deleted: When a DELETE statement is completed, the number of rows deleted is returned in the ROW_COUNT condition area item in the SQL Diagnostics Area (or SQLERRD(3) in the SQLCA). The value in the ROW_COUNT item does not include the number of rows that were deleted as a result of a CASCADE delete rule or a trigger.

For a description of the SQLCA, see Appendix C, "SQLCA (SQL communication area)," on page 1513.

DELETE Performance: An SQL DELETE statement that does not contain a WHERE clause will delete all rows of a table. In this case, the rows may be deleted using either a clear operation (if not running under commitment control) or a change file operation (if running under commitment control). If running under commitment control, the deletes can still be committed or rolled back. This implementation will be much faster than individually deleting each row, but individual journal entries for each row will not be recorded in the journal. This technique will only be used if all the following are true:

- The target table is not a view.
- A significant number of rows are being deleted.
- The job issuing the DELETE statement does not have an open cursor on the file (not including pseudo-closed SQL cursors).
- No other job has a lock on the table.
- The table does not have an active delete trigger.
- The table is not the parent in a referential constraint with a CASCADE, SET NULL, or SET DEFAULT delete rule.
- The user issuing the DELETE statement has *OBJMGT or *OBJALTER system authority on the table in addition to the DELETE privilege.

If this technique is successful, the number of increments (see the SIZE keyword on the CHGPF CL command) is set to zero.

Referential integrity considerations: The DB2_ROW_COUNT_SECONDARY condition information item in the SQL Diagnostics Area (or SQLERRD(5) in the SQLCA) shows the number of rows affected by referential constraints. It includes rows that were deleted as the result of a CASCADE delete rule and rows in which foreign keys were set to NULL or the default value as the result of a SET NULL or SET DEFAULT delete rule.

For a description of DB2_ROW_COUNT_SECONDARY, see "GET DIAGNOSTICS" on page 1194. For a description of the SQLCA, see Appendix C, "SQLCA (SQL communication area)," on page 1513.

REXX: Variables cannot be used in the DELETE statement within a REXX procedure. Instead, the DELETE must be the object of a PREPARE and EXECUTE using parameter markers.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword NONE can be used as a synonym for NC.
- The keyword CHG can be used as a synonym for UR.
- The keyword ALL can be used as a synonym for RS.
- The FROM keyword is optional.

Examples

Example 1: Delete department (DEPTNO) 'D11' from the DEPARTMENT table.

```
DELETE FROM DEPARTMENT
  WHERE DEPTNO = 'D11'
```

Example 2: Delete all the departments from the DEPARTMENT table (that is, empty the table).

```
DELETE FROM DEPARTMENT
```

Example 3: Use a Java program statement to delete all the subprojects (MAJPROJ is NULL) from the PROJECT table on the connection context 'ctx', for a department (DEPTNO) equal to that in the host variable HOSTDEPT (java.lang.String).

```
#sql [ctx] { DELETE FROM PROJECT
               WHERE DEPTNO = :HOSTDEPT
               AND MAJPROJ IS NULL };
```

Example 4: Code a portion of a Java program that will be used to display retired employees (JOB) and then, if requested to do so, remove certain employees from the EMPLOYEE table on the connection context 'ctx'.

```
#sql iterator empIterator implements sqlj.runtime.ForUpdate
     ( ... );
empIterator C1;

#sql [ctx] C1 = { SELECT * FROM EMPLOYEE
                    WHERE JOB = 'RETIRED' };

#sql {  FETCH C1 INTO ...     };
while ( !C1.endFetch() )  {
   System.out.println( ... );
                ...
   if ( condition for deleting row ) {
       #sql [ctx] { DELETE FROM EMPLOYEE
                      WHERE CURRENT OF C1  };
   }
   #sql { FETCH C1 INTO ...     };
}
C1.close();
```

DESCRIBE

The DESCRIBE statement obtains information about a prepared statement.

For an explanation of prepared statements, see "PREPARE" on page 1293.

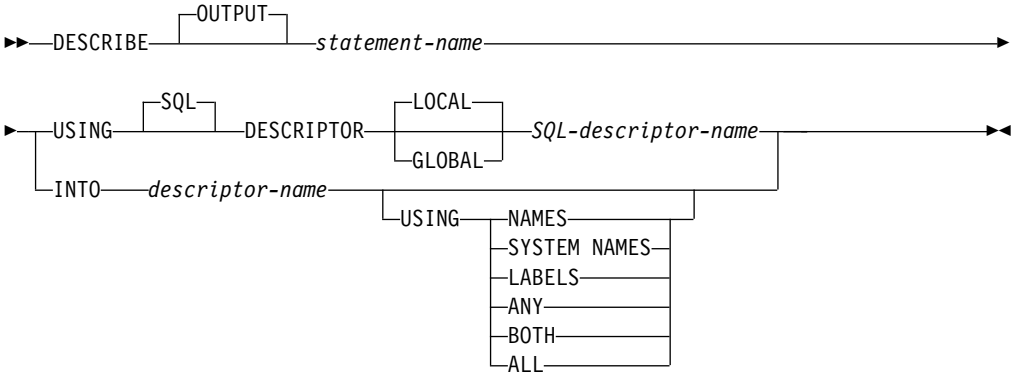
Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required. See "PREPARE" on page 1293 for the authorization required to create a prepared statement.

Syntax



Description

statement-name
Identifies the prepared statement. When the DESCRIBE statement is executed, the name must identify a prepared statement at the application server.

If the prepared statement is a fullselect or VALUES INTO statement, the information returned describes the columns in its result table. If the prepared statement is a CALL statement, the information returned describes the OUT and INOUT parameters of the procedure.

USING
Identifies an SQL descriptor.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "GET DESCRIPTOR" on page 1182 for an explanation of the information that is placed in the SQL descriptor.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, "SQLDA (SQL descriptor area)," on page 1523. Before the DESCRIBE statement is executed, the following variable in the SQLDA must be set.

SQLN Indicates the number of SQLVAR entries provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the DESCRIBE statement is executed. For information about techniques to determine the number of occurrences requires, see "Determining how many SQLVAR occurrences are needed" on page 1526.

The rules for REXX are different. For more information, see the Embedded SQL Programming topic collection.

When the DESCRIBE statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID
The first 6 bytes are set to 'SQLDA ' (that is, 5 letters followed by the space character).

The seventh byte is set based on the result columns described:

- If the SQLDA contains two, three, or four SQLVAR entries for every select list item (or, column of the result table), the seventh byte is set to '2', '3', or '4'. This technique is used in order to accommodate LOB or distinct type result columns, labels, and system names.
- Otherwise, the seventh byte is set to the space character.

The seventh byte is set to the space character if there is not enough room in the SQLDA to contain the description of all result columns.

The eighth byte is set to the space character.

SQLDABC
Length of the SQLDA in bytes.

SQLD If the prepared statement is a SELECT, SQLD is set to the number of columns in its result table plus the number of extended SQLVAR entries. For information about extended SQLVAR entries see, "Field descriptions in an occurrence of SQLVAR" on page 1528. If the prepared statement is a CALL statement, SQLD is set to the number of OUT and INOUT parameters of the procedure. If the prepared statement is a VALUES INTO, SQLD is set to the number of expressions in the VALUES cause plus the number of extended SQLVAR entries. Otherwise, SQLD is set to 0.

SQLVAR
If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column of the result table (or parameter or expression in the VALUES clause), the second occurrence of SQLVAR contains a description of the second column of the result table (or parameter or expression in the

VALUES clause), and so on. For information about the values assigned to SQLVAR occurrences, see "Field descriptions in an occurrence of SQLVAR" on page 1528.

USING
Specifies what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist or if the length of a name is greater than 30, SQLNAME is set to a length of 0.

NAMES
Assigns the name of the column (or parameter). This is the default. For the DESCRIBE of a prepared statement where the name is explicitly listed in the select-list, the name specified is returned. The column name returned is case sensitive and without delimiters.

SYSTEM NAMES
Assigns the system column name of the column.

LABELS
Assigns the label of the column. (Column labels are defined by the LABEL statement.) Only the first 20 bytes of the label are returned.

ANY
Assigns the column label. If the column has no label, the column name is used instead.

BOTH
Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 2*n* or 3*n*(where *n* is the number of columns in the table or view). The first *n* occurrences of SQLVAR contain the column names. Either the second or third *n* occurrences contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

ALL
Assigns the label, column name, and system column name. In this case three or four occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 3*n* or 4*n* (where *n* is the number of columns in the result table). The first *n* occurrences of SQLVAR contain the system column names. The second or third *n* occurrences contain the column labels. The third or fourth *n* occurrences contain the column names if they are different from the system column name. Otherwise the SQLNAME field is set to a length of zero. If there are no distinct types, the labels are returned in the second set of SQLVAR entries and the column names are returned in the third set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries and the column names are returned in the fourth set of SQLVAR entries.

Notes

PREPARE INTO: Information about a prepared statement can also be obtained by using the INTO clause of the PREPARE statement.

Allocating the SQL descriptor: Before the DESCRIBE statement is executed, the SQL descriptor must be allocated using the ALLOCATE DESCRIPTOR statement. If the number of descriptor items allocated is less than the number of result columns, a warning (SQLSTATE 01005) is returned.

Allocating the SQLDA: In C, COBOL, PL/I, and RPG, before the DESCRIBE or PREPARE INTO statement is executed, enough storage must be allocated for some number of SQLVAR occurrences. SQLN must then be set to the number of SQLVAR occurrences that were allocated. To obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR entries must not be less than the number of columns. Furthermore, if the columns include LOBs or distinct types, the number of occurrences of SQLVAR entries should be two times the number of columns. See "Determining how many SQLVAR occurrences are needed" on page 1526 for more information. Among the possible ways to allocate the SQLDA are the three described below:

First technique

Allocate an SQLDA with enough occurrences of SQLVAR entries to accommodate any select list that the application will have to process. At the extreme, the number of SQLVARs could equal two times the maximum number of columns allowed in a result table. Having done the allocation, the application can use this SQLDA repeatedly.

This technique uses a large amount of storage that is never deallocated, even when most of this storage is not used for a particular select list.

Second technique

Repeat the following three steps for every processed select list:

1. Execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR entries; that is, an SQLDA for which SQLN is zero. The value returned for SQLD is either the required number of occurrences of SQLVAR entries or the number of result columns. Because there were no SQLVAR entries, a warning will be issued.[97]
2. If the seventh byte of SQLDAID field is not a blank, then allocate an SQLDA with (the value in the seventh byte of SQLDAID) * SQLD occurrences and set SQLN in the new SQLDA to (the value in the seventh byte of SQLDAID) * SQLD. Otherwise, allocate an SQLDA with SQLD occurrences and set SQLN in the new SQLDA to the value of SQLD.
3. Execute the DESCRIBE statement again, using this new SQLDA.

This technique allows better storage management than the first technique, but it doubles the number of DESCRIBE statements.

Third technique

Allocate an SQLDA that is large enough to handle most, and perhaps all, select lists but is also reasonably small. If an execution of DESCRIBE fails because the SQLDA is too small, allocate a larger SQLDA and execute DESCRIBE again. For the new SQLDA, use the value of SQLD (or double the value of SQLD) returned from the first execution of DESCRIBE for the number of occurrences of SQLVAR entries.

This technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

97. If LOBs or UDTs are not in the result set, the warning is only returned if the standards option is specified. For information about the standards option, see "Standards compliance" on page xi.

Considerations for implicitly hidden columns: A DESCRIBE OUTPUT statement only returns information about implicitly hidden columns if the column (of a base table that is defined as implicitly hidden) is explicitly specified as part of the SELECT list of the final result table of the query described. If implicitly hidden columns are not part of the result table of a query, a DESCRIBE OUTPUT statement that returns information about that query will not contain information about any implicitly hidden columns.

Example

In a C program, execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR entries. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR entrires and then execute a DESCRIBE statement using that SQLDA.

```
EXEC SQL  BEGIN DECLARE SECTION;
  char stmt1_str [200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;
struct sqlda initialsqlda;
struct sqlda *sqldaPtr;

EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

... /* code to prompt user for a query, then to generate    */
    /* a select-statement in the stmt1_str                  */
EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;

... /* code to set SQLN to zero and to allocate the SQLDA   */
EXEC SQL  DESCRIBE STMT1_NAME INTO :initialsqlda;

if (initialsqlda.sqld == 0);  /* statement is a select-statement */
  {
   ... /* Code to allocate correct size SQLDA (sets sqldaPtr) */

   if (strcmp(SQLSTATE,"01005") == 0)
     {
      sqldaPtr->sqln = 2*initialsqlda.sqld;
      SETSQLDOUBLED(sqldaPtr, SQLDOUBLED);
     }
   else
     {
      sqldaPtr->sqln = initialsqlda.sqld;
      SETSQLDOUBLED(sqldaPtr, SQLSINGLED);
     }
   EXEC SQL  DESCRIBE STMT1_NAME INTO :*sqldaPtr;

   ... /* code to prepare for the use of the SQLDA             */
   EXEC SQL  OPEN DYN_CURSOR;

   ... /* loop to fetch rows from result table                 */
   EXEC SQL  FETCH DYN_CURSOR USING DESCRIPTOR :*sqldaPtr;

   ...
  }
...
```

DESCRIBE CURSOR

The DESCRIBE CURSOR statement gets information about a cursor. The information, such as column information, is put into a descriptor.

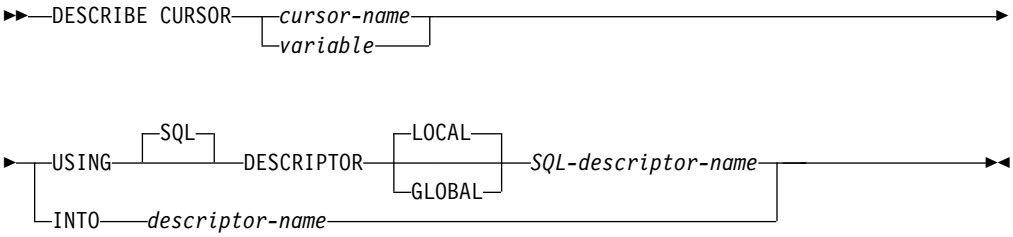
Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

cursor-name **or** *variable*
Identifies a cursor that has already been open or allocated in the source program.

If a *variable* is specified:

- It must be a character-string variable or Unicode graphic-string. It cannot be a global variable.
- It must not be followed by an indicator variable.
- The cursor name that is contained within the variable must be left-justified and must be padded on the right with blanks if its length is less than that of the variable.
- The name of the cursor must be in uppercase unless the cursor name is a delimited name.

USING
Identifies an SQL descriptor.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

The information returned in the descriptor area describes the columns in the result set associated with the named cursor. After the DESCRIBE CURSOR is executed, the contents of the descriptor area are the same as after a DESCRIBE of a SELECT with the following addition.

- DB2_CURSOR_HOLD can be returned from the GET DESCRIPTOR statement to indicate whether the cursor was declared WITH HOLD in the procedure.

See "GET DESCRIPTOR" on page 1182 for an explanation of the information that is placed in the SQL descriptor.

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, "SQLDA (SQL descriptor area)," on page 1523. Before the DESCRIBE CURSOR statement is executed, the following variable in the SQLDA must be set.

SQLN Specifies the number of SQLVAR occurrences provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the DESCRIBE CURSOR statement is executed. For information about techniques to determine the number of occurrences required, see "Determining how many SQLVAR occurrences are needed" on page 1526.

When the DESCRIBE statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID
The first 5 bytes are set to 'SQLRS'. Bytes 6 to 8 are reserved. If the cursor is declared WITH HOLD in the procedure, the high-order bit of the 8th byte is set to 1.

SQLDABC
Length of the SQLDA in bytes.

SQLD The number of columns in the result table plus the number of extended SQLVAR entries. For information about extended SQLVAR entries see, "Field descriptions in an occurrence of SQLVAR" on page 1528.

SQLVAR
If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column of the result table, the second occurrence of SQLVAR contains a description of the second column of the result table, and so on. For information about the values assigned to SQLVAR occurrences, see "Field descriptions in an occurrence of SQLVAR" on page 1528.

Notes

Allocating the SQL descriptor: Before the DESCRIBE CURSOR statement is executed, the SQL descriptor must be allocated using the ALLOCATE DESCRIPTOR statement. If the number of descriptor items allocated is less than the number of columns in the cursor result set, a warning (SQLSTATE 01005) is returned.

Allocating the SQLDA: Before the DESCRIBE CURSOR statement is executed, the value of SQLN must be set to a value greater than or equal to zero to indicate how many occurrences of SQLVAR are provided in the SQLDA and enough storage must be allocated to contain SQLN occurrences. To obtain the description of the columns of the cursor result set, the number of occurrences of SQLVAR must not be less than the number of columns.

For a description of techniques that can be used to allocate the SQLDA, see Appendix D, "SQLDA (SQL descriptor area)," on page 1523.

Example

Place information about the result set associated with cursor C1 into an SQL descriptor.

```
EXEC SQL ALLOCATE DESCRIPTOR 'DESCR1';
EXEC SQL DESCRIBE CURSOR C1 USING SQL DESCRIPTOR 'DESCR1';
```

DESCRIBE INPUT

The DESCRIBE INPUT statement obtains information about the IN and INOUT parameter markers of a prepared statement.

For an explanation of prepared statements, see "PREPARE" on page 1293.

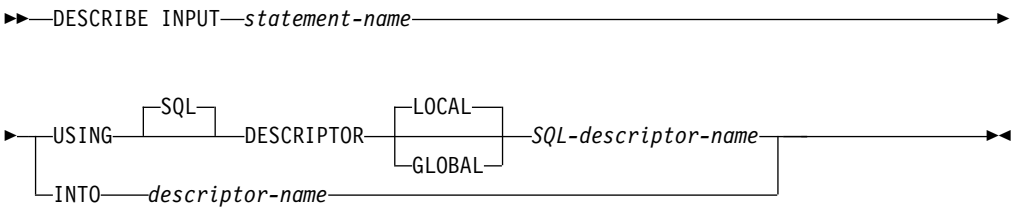
Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

None required. See "PREPARE" on page 1293 for the authorization required to create a prepared statement.

Syntax



Description

statement-name
Identifies the prepared statement. When the DESCRIBE INPUT statement is executed, the name must identify a prepared statement at the current server.

USING
Identifies an SQL descriptor.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "GET DESCRIPTOR" on page 1182 for an explanation of the information that is placed in the SQL descriptor.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, "SQLDA (SQL descriptor area)," on page 1523. Before the DESCRIBE INPUT statement is executed, the following variable in the SQLDA must be set.

SQLN Specifies the number of SQLVAR occurrences provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the

DESCRIBE INPUT statement is executed. For information about techniques to determine the number of occurrences requires, see "Determining how many SQLVAR occurrences are needed" on page 1526.

When the DESCRIBE INPUT statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID
The first 6 bytes are set to 'SQLDA ' (that is, 5 letters followed by the space character).

The seventh byte is set based on the parameter markers described:

- If the SQLDA contains two SQLVAR entries for every input parameter marker, the seventh byte is set to '2'. This technique is used in order to accommodate LOB input parameters.
- Otherwise, the seventh byte is set to the space character.

The seventh byte is set to the space character if there is not enough room in the SQLDA to contain the description of all input parameter markers.

The eighth byte is set to the space character.

SQLDABC
Length of the SQLDA in bytes.

SQLD The number of input parameter markers in the prepared statement.

SQLVAR
If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first input parameter marker, the second occurrence of SQLVAR contains a description of the second input parameter marker, and so on. For information about the values assigned to SQLVAR occurrences, see "Field descriptions in an occurrence of SQLVAR" on page 1528.

Notes

Allocating the SQL descriptor: Before the DESCRIBE INPUT statement is executed, the SQL descriptor must be allocated using the ALLOCATE DESCRIPTOR statement. The number of descriptor items allocated must not be less than the number of input parameter markers or an error is returned.

Allocating the SQLDA: Before the DESCRIBE INPUT statement is executed, enough storage must be allocated for some number of SQLVAR occurrences. SQLN must then be set to the number of SQLVAR occurrences that were allocated. To obtain the description of the input parameter markers in the prepared statement, the number of occurrences of SQLVAR must not be less than the number of input parameter markers. Furthermore, if the input parameter markers include LOBs or distinct types, the number of occurrences of SQLVAR should be two times the number of input parameter markers. See "Determining how many SQLVAR occurrences are needed" on page 1526 for more information.

If not enough occurrences are provided to return all sets of occurrences, SQLN is set to the total number of occurrences necessary to return all information. Otherwise, SQLN is set to the number of input parameter markers.

Among the possible ways to allocate the SQLDA are the three described below:

First technique

Allocate an SQLDA with enough occurrences of SQLVAR entries to accommodate any number of input parameter markers that the application will have to process. At the extreme, the number of SQLVARs could equal two times the maximum number of parameter markers allowed in a prepared statement. Having done the allocation, the application can use this SQLDA repeatedly.

This technique uses a large amount of storage that is never deallocated, even when most of this storage is not used for a particular prepared statement.

Second technique

Repeat the following three steps for every processed prepared statement:

1. Execute a DESCRIBE INPUT statement with an SQLDA that has no occurrences of SQLVAR entries, that is, an SQLDA for which SQLN is zero. The value returned for SQLD is the number of input parameter markers in the prepared statement. This value is either the required number of occurrences of SQLVAR entries or half the required number. Because there were no SQLVAR entries, a warning will be issued. [98]
2. If the SQLSTATE accompanying that warning is equal to 01005, allocate an SQLDA with 2 * SQLD occurrences and set SQLN in the new SQLDA to 2 * SQLD. Otherwise, allocate an SQLDA with SQLD occurrences and set SQLN in the new SQLDA to the value of SQLD.
3. Execute the DESCRIBE INPUT statement again, using this new SQLDA.

This technique allows better storage management than the first technique, but it doubles the number of DESCRIBE INPUT statements.

Third technique

Allocate an SQLDA that is large enough to handle most, and perhaps all, parameter markers in prepared statements but is also reasonably small. If an execution of DESCRIBE INPUT fails because the SQLDA is too small, allocate a larger SQLDA and execute DESCRIBE INPUT again. For the new SQLDA, use the value of SQLD (or double the value of SQLD) returned from the first execution of DESCRIBE INPUT for the number of occurrences of SQLVAR entries.

This technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

Examples

Example 1: In a C program, execute a DESCRIBE INPUT statement with an SQLDA that has enough to describe any number of input parameter markers a prepared statement might have. Assume that five parameter markers at most will need to be described and that the input data does not contain LOBs.

98. If LOBs or UDTs are not in the result set, the warning is only returned if the standards option is specified. For information about the standards option, see "Standards compliance" on page xi.

```
EXEC SQL  BEGIN DECLARE SECTION;
  char stmt1_str [200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;
struct sqlda initialsqlda;
struct sqlda *sqldaPtr;

... /* stmt1_str contains INSERT statement with VALUES     */
    /* clause                                               */
EXEC SQL  PREPARE STMT1_NAME   FROM :stmt1_str;

... /* code to set SQLN to five and to allocate the SQLDA */
EXEC SQL  DESCRIBE INPUT  STMT1_NAME INTO  :SQLDA;

 ...
```

Example 2: Allocate a descriptor called 'NEWDA' large enough to hold 20 item descriptor areas and use it on DESCRIBE INPUT.

```
EXEC SQL ALLOCATE DESCRIPTOR 'NEWDA'
  WITH MAX 20;

EXEC SQL DESCRIBE INPUT STMT1
  USING SQL DESCRIPTOR 'NEWDA';
```

DESCRIBE PROCEDURE

The DESCRIBE PROCEDURE statement gets information about the result sets returned by a procedure. The information, such as the number of result sets, is put into a descriptor.

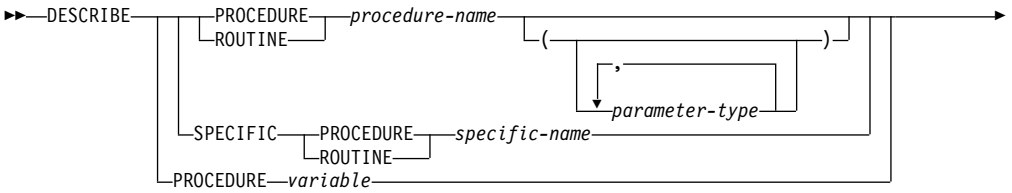
Invocation

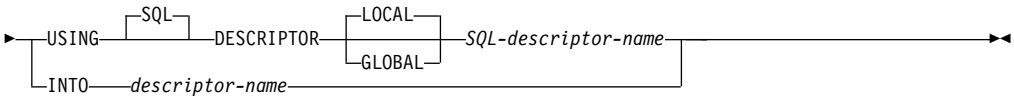
This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

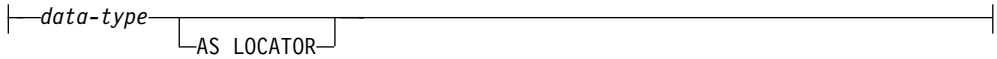
None required.

Syntax

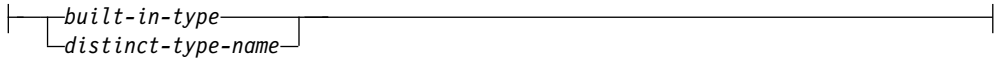




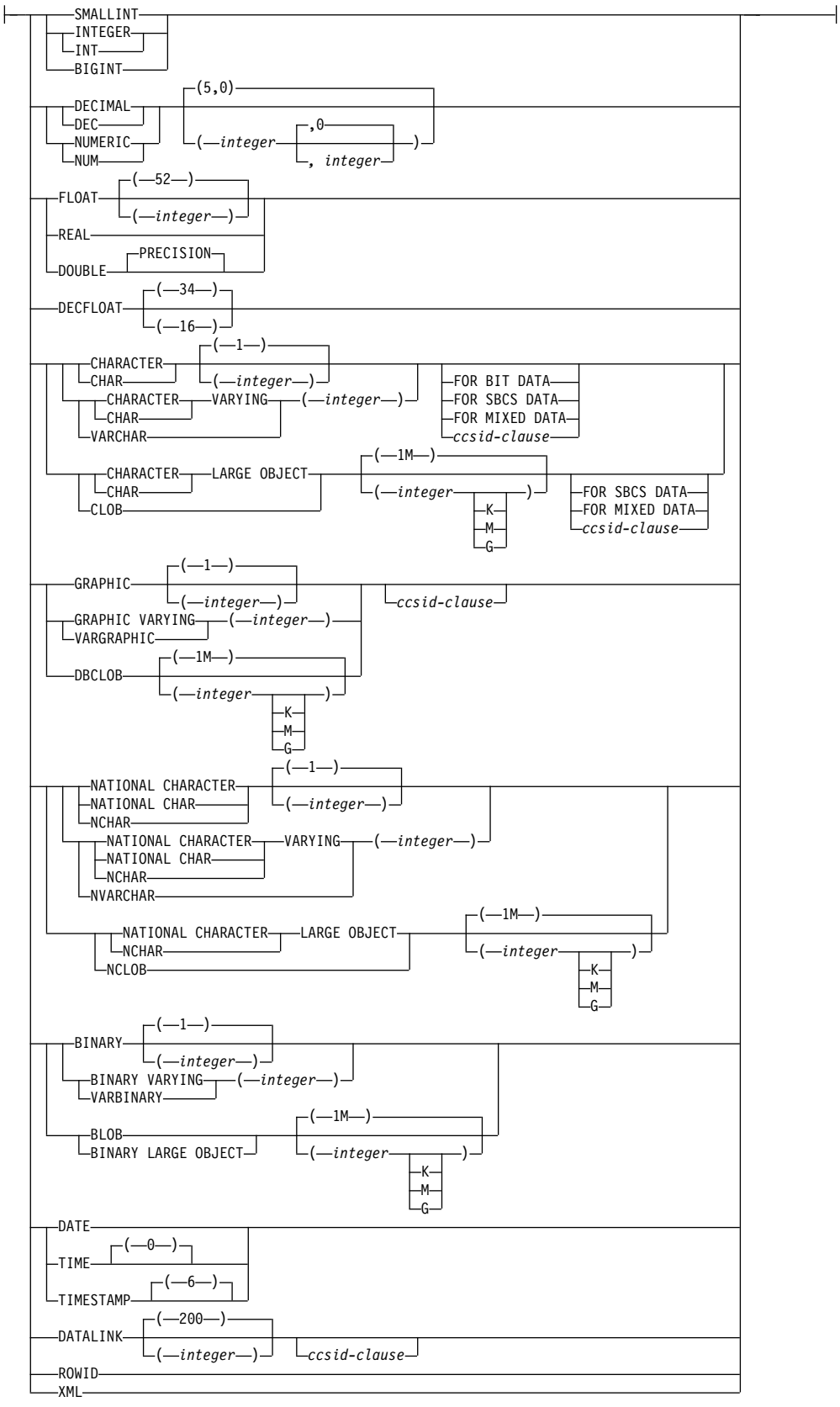
parameter-type:



data-type:



built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(integer
,0
, integer
)
FLOAT
(52)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
FOR BIT DATA
FOR SBCS DATA
FOR MIXED DATA
ccsid-clause
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
K
M
G
GRAPHIC
GRAPHIC VARYING
VARGRAPHIC
DBCLOB
NATIONAL CHARACTER
NATIONAL CHAR
NCHAR
NATIONAL CHARACTER VARYING
NATIONAL CHAR VARYING
NVARCHAR
NATIONAL CHARACTER LARGE OBJECT
NCHAR LARGE OBJECT
NCLOB
BINARY
BINARY VARYING
VARBINARY
BLOB
BINARY LARGE OBJECT
DATE
TIME
TIMESTAMP
(0)
(6)
DATALINK
(200)
ROWID
XML

ccsid-clause:

Description

procedure-name **or** *specific-name* **or** *variable*
Identifies the procedure that returned one or more result sets. When the DESCRIBE PROCEDURE statement is executed, the procedure name must identify a procedure that the requester has already invoked using the SQL CALL statement.

PROCEDURE or SPECIFIC PROCEDURE
Identifies the procedure to be described. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*
Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be labeled on. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name
Identifies the name of the procedure.

(parameter-type, ...)
Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have

to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).

- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

variable
Specifies a variable that contains a procedure or specific name. If *variable* is specified:

- It must be a character-string variable or Unicode graphic-string variable. It cannot be a global variable.
- It must not be followed by an indicator variable.
- The name that is contained within the variable must be left-justified and must be padded on the right with blanks if its length is less than that of the variable.
- The name must be in uppercase unless it is a delimited name.

If only one procedure with this name has been invoked using the CALL statement, the variable is used as a procedure name. If multiple procedures with this name have been invoked, the variable is used as a specific name.

USING
Identifies an SQL descriptor.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

After the DESCRIBE PROCEDURE is executed, the following values can be retrieved with the GET DESCRIPTOR statement:

- DB2_RESULT_SETS_COUNT contains the total number of result sets. A value of 0 indicates there are no result sets.

There is one descriptor area item for each result set:

- DB2_RESULT_SET_LOCATOR contains the result set locator value associated with the result set.
- DB2_CURSOR_NAME contains the name of the cursor used by the procedure to return the result set.
- DB2_RESULT_SET_ROWS contains the estimated number of rows in the result set. This is set to -1 if the number is unknown.

See "GET DESCRIPTOR" on page 1182 for an explanation of the information that is placed in the SQL descriptor.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, "SQLDA (SQL descriptor area)," on page 1523. Before the DESCRIBE PROCEDURE statement is executed, the following variable in the SQLDA must be set.

SQLN Specifies the number of SQLVAR occurrences provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the DESCRIBE PROCEDURE statement is executed.

When the DESCRIBE statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID
The first 5 bytes are set to 'SQLPR'. Bytes 6 to 8 are reserved.

SQLDABC
Length of the SQLDA in bytes.

SQLD The total number of result sets. A value of 0 indicates there are no result sets.

SQLVAR
If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first result set, the second occurrence of SQLVAR contains a description of the second result set, and so on. For each SQLVAR entry:

- The SQLDATA field is set to the result set locator value associated with the result set.
- The SQLIND field is set to the estimated number of rows in the result set. This is set to -1 if the number is unknown.
- The SQLNAME field is set to the name of the cursor used by the stored procedure to return the result set. The cursor name is truncated to 30 characters if it is longer than 30 characters.

Notes

DESCRIBE PROCEDURE does not return information about the parameters expected by the procedure.

The CALL to the procedure must precede the DESCRIBE PROCEDURE statement.

Allocating the SQL descriptor: Before the DESCRIBE PROCEDURE statement is executed, the SQL descriptor must be allocated using the ALLOCATE DESCRIPTOR statement. If the number of descriptor items allocated is less than the number of result sets for the procedure, a warning (SQLSTATE 01005) is returned.

Allocating the SQLDA: Before the DESCRIBE PROCEDURE statement is executed, the value of SQLN must be set to a value greater than or equal to zero to indicate how many occurrences of SQLVAR are provided in the SQLDA and enough storage must be allocated to contain SQLN occurrences. To obtain the description of the result sets for the procedure, the number of occurrences of SQLVAR must not be less than the number result sets.

If not enough occurrences are provided to return all sets of occurrences, SQLN is set to the total number of occurrences necessary to return all information. Otherwise, SQLN is set to the number of result sets.

Assignment of locator values: If a SET RESULT SETS statement was executed in the procedure, the SET RESULT SETS statement identifies the result sets. The locator values are assigned to the items in the descriptor area or the SQLVAR entries in the SQLDA in the order specified on the SET RESULT SETS statement. If a SET RESULT SETS statement was not executed in the procedure, locator values are assigned to the items in the descriptor area or to the SQLVAR entries in the SQLDA in the order that the associated cursors are opened at run time. Locator values are not provided for cursors that are closed when control is returned to the invoking application. If a cursor was closed and later re-opened before returning to the invoking application, the most recently executed OPEN CURSOR statement for the cursor is used to determine the order in which the locator values are returned for the procedure result sets. For example, assume procedure P1 opens three cursors A, B, and C, closes cursor B, and then issues another OPEN CURSOR statement for cursor B before returning to the invoking application. The locator values are assigned in the order A, C, B.

Alternatively, an ASSOCIATE LOCATORS statement can be used to copy the locator values to result set locator variables.

Example

Place information about the result sets returned by procedure P1 into an SQL descriptor:

```
EXEC SQL CALL P1;
EXEC SQL ALLOCATE DESCRIPTOR 'DESC1';
EXEC SQL DESCRIBE PROCEDURE P1 USING SQL DESCRIPTOR 'DESC1';
```

DESCRIBE TABLE

The DESCRIBE TABLE statement obtains information about a table or view.

Invocation

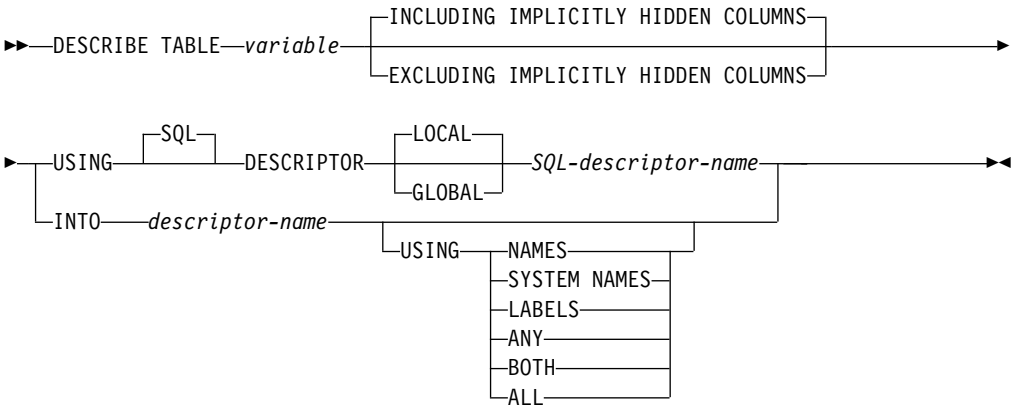
This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table or view identified in the statement:
  - The system authority of *OBJOPR on the table or view
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

Syntax



Description

variable
Identifies the table or view to describe. When the DESCRIBE TABLE statement is executed:

- The name must identify a table or view that exists at the application server.
- The *variable* must be a character-string or Unicode graphic-string variable and must not include an indicator variable. It cannot be a global variable.
- The table name that is contained within the *variable* must be left-justified and must be padded on the right with blanks if its length is less than that of the *variable*.
- The name of the table must be in uppercase unless it is a delimited name.

INCLUDING IMPLICITLY HIDDEN COLUMNS or EXCLUDING IMPLICITLY HIDDEN COLUMNS
Specifies whether information should be returned for implicitly hidden column in a table.

INCLUDING IMPLICITLY HIDDEN COLUMNS
Specifies that information is returned for columns defined as implicitly hidden. This is the default.

EXCLUDING IMPLICITLY HIDDEN COLUMNS
Specifies that information is not returned for columns defined as implicitly hidden.

When the DESCRIBE TABLE statement is executed, the database manager assigns values to the variables of the SQL descriptor or SQLDA as follows:

USING
Identifies an SQL descriptor.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "GET DESCRIPTOR" on page 1182 for an explanation of the information that is placed in the SQL descriptor.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, "SQLDA (SQL descriptor area)," on page 1523. Before the DESCRIBE TABLE statement is executed, the following variable in the SQLDA must be set.

SQLN Specifies the number of SQLVAR occurrences provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the DESCRIBE TABLE statement is executed. For information about techniques to determine the number of occurrences requires, see "Determining how many SQLVAR occurrences are needed" on page 1526.

The rules for REXX are different. For more information, see the Embedded SQL Programming topic collection.

When the DESCRIBE statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID
The first 6 bytes are set to 'SQLDA ' (that is, 5 letters followed by the space character).

The seventh byte is set based on the column described:

- If the SQLDA contains two, three, or four SQLVAR entries for every column of the table, the seventh byte is set to '2', '3', or '4'. This technique is used in order to accommodate LOB or distinct type result columns, labels, and system names.
- Otherwise, the seventh byte is set to the space character.

The seventh byte is set to the space character if there is not enough room in the SQLDA to contain the description of all columns.

The eighth byte is set to the space character.

SQLDABC
Length of the SQLDA in bytes.

SQLD The number of columns in the table plus the number of extended SQLVAR entries. For information about extended SQLVAR entries see, "Field descriptions in an occurrence of SQLVAR" on page 1528.

SQLVAR
If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column of the table, the second occurrence of SQLVAR contains a description of the second column of the table, and so on. For information about the values assigned to SQLVAR occurrences, see "Field descriptions in an occurrence of SQLVAR" on page 1528.

USING
Specifies what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist or if the length of a name is greater than 30, SQLNAME is set to a length of 0.

NAMES
Assigns the name of the column. The column name returned is case sensitive and without delimiters. This is the default.

SYSTEM NAMES
Assigns the system column name of the column.

LABELS
Assigns the label of the column. (Column labels are defined by the LABEL statement.) Only the first 20 bytes of the label are returned.

ANY
Assigns the column label. If the column has no label, the column name is used instead.

BOTH
Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the table contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 2*n* or 3*n*(where *n* is the number of columns in the table or view). The first *n* occurrences of SQLVAR contain the column names if they are different from the system column name. Either the second or third *n* occurrences contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

ALL
Assigns the label, column name, and system column name. In this case three or four occurrences of SQLVAR per column, depending on whether the table contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 3*n* or 4*n* (where *n* is the number of columns in the table). The first *n* occurrences of SQLVAR contain the system column

names. The second or third *n* occurrences contain the column labels. The third or fourth *n* occurrences contain the column names. If there are no distinct types, the labels are returned in the second set of SQLVAR entries and the column names are returned in the third set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries and the column names are returned in the fourth set of SQLVAR entries.

Notes

Allocating the SQL descriptor: Before the DESCRIBE TABLE statement is executed, the SQL descriptor must be allocated using the ALLOCATE DESCRIPTOR statement. If the number of descriptor items allocated is less than the number of columns in the table or view, a warning (SQLSTATE 01005) is returned.

Allocating the SQLDA: Before the DESCRIBE TABLE statement is executed, the value of SQLN must be set to a value greater than or equal to zero to indicate how many occurrences of SQLVAR are provided in the SQLDA and enough storage must be allocated to contain SQLN occurrences. To obtain the description of the columns of the table or view, the number of occurrences of SQLVAR must not be less than the number of columns. Furthermore, if USING BOTH or USING ALL is specified, or if the columns include LOBs or distinct types, the number of occurrences of SQLVAR should be two, three, or four times the number of columns. See "Determining how many SQLVAR occurrences are needed" on page 1526 for more information.

If not enough occurrences are provided to return all sets of occurrences, SQLN is set to the total number of occurrences necessary to return all information. Otherwise, SQLN is set to the number of columns.

For a description of techniques that can be used to allocate the SQLDA, see Appendix D, "SQLDA (SQL descriptor area)," on page 1523.

Example

In a C program, execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR and then execute a DESCRIBE statement using that SQLDA.

```
EXEC SQL  BEGIN DECLARE SECTION;
  char table_name[201];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

.../*code to prompt user for a table or view */
.../*code to set SQLN to zero and to allocate the SQLDA */
EXEC SQL  DESCRIBE TABLE :table_name INTO :sqlda;

... /* code to check that SQLD is greater than zero, to set */
    /* SQLN to SQLD, then to re-allocate the SQLDA          */
EXEC SQL  DESCRIBE TABLE :table_name INTO :sqlda;

.
.
.
```

DISCONNECT

The DISCONNECT statement ends one or more connections for unprotected conversations.

Invocation

This statement can only be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

DISCONNECT is not allowed in a trigger or function.

Authorization

If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax

Description

server-name **or** *variable*

Identifies the application server by the specified server name or the server name contained in the variable. It can be a global variable if it is qualified with schema name. If a variable is specified:

- It must be a character-string variable.
- It must not be followed by an indicator variable
- The server name must be left-justified within the variable and must conform to the rules for forming an ordinary identifier
- If the length of the server name is less than the length of the variable, it must be padded on the right with blanks.

When the DISCONNECT statement is executed, the specified server name or server name contained in the variable must identify an existing dormant or current connection of the activation group. The identified connection cannot use a protected conversation.

CURRENT

Identifies the current connection of the activation group. The activation group must be in the connected state. The current connection must not use a protected conversation.

ALL or ALL SQL

Identifies all existing connections of the activation group (local as well as

remote connections). An error or warning does not occur if no connections exist when the statement is executed. None of the connections can use protected conversations.

Notes

DISCONNECT and CONNECT (Type 1): Using CONNECT (Type 1) semantics does not prevent using DISCONNECT.

Connection restrictions: An identified connection must not be a connection that was used to execute SQL statements during the current unit of work and must not be a connection for a protected conversation. To end connections on protected conversations, use the RELEASE statement. Local connections are never considered to be protected conversations.

The DISCONNECT statement should be executed immediately after a commit operation. If DISCONNECT is used to end the current connection, the next executed SQL statement must be CONNECT or SET CONNECTION.

ROLLBACK does not reconnect a connection that has been ended by DISCONNECT.

Successful disconnect: If the DISCONNECT statement is successful, each identified connection is ended. If the current connection is destroyed, the activation group is placed in the unconnected state.

DISCONNECT closes cursors, releases resources, and prevents further use of the connection.

DISCONNECT ALL ends the connection to the local application server. A connection is ended even though it has an open cursor defined with the WITH HOLD clause.

Unsuccessful disconnect: If the DISCONNECT statement is unsuccessful, the connection state of the activation group and the states of its connections are unchanged.

Resource considerations for remote connections: Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be ended as soon as possible and a remote connection that is going to be reused should not be destroyed.

Examples

Example 1: The connection to TOROLAB1 is no longer needed. The following statement is executed after a commit operation.

```
EXEC SQL DISCONNECT TOROLAB1;
```

Example 2: The current connection is no longer needed. The following statement is executed after a commit operation.

```
EXEC SQL DISCONNECT CURRENT;
```

Example 3: The existing connections are no longer needed. The following statement is executed after a commit operation.

```
EXEC SQL DISCONNECT ALL;
```

DROP

The DROP statement drops an object. Objects that are directly or indirectly dependent on that object may also be dropped.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

To drop a table, view, index, alias, or package, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authorities of *OBJOPR and *OBJEXIST on the object to be dropped
  - If the object is a table or view, the system authorities of *OBJOPR and *OBJEXIST on any views, indexes, and logical files that are dependent on that table or view
  - The system authority *EXECUTE on the library that contains the object to be dropped
- Administrative authority

To drop a schema, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authorities of *OBJEXIST, *OBJOPR, *EXECUTE, and *READ on the library to be dropped.
  - The system authorities of *OBJOPR and *OBJEXIST on all objects in the schema and *OBJOPR and *OBJEXIST on any views, indexes and logical files that are dependent on tables and views in the schema.
  - Any additional authorities required to delete other object types that exist in the schema. For example, *OBJMGT to the data dictionary if the schema contains a data dictionary, and some system data authority to the journal receiver. For more information, see Security Reference.
- Administrative authority

To drop a user-defined type, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authorities of *OBJOPR and *OBJEXIST on the type to be dropped
  - The system authority *EXECUTE on the library that contains the type to be dropped
  - The DELETE privilege on the SYSTYPES, SYSPARMS, and SYSROUTINES catalog tables, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To drop a global variable, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:

  - The system authority of *OBJEXIST on the *SRVPGM object for the global variable to be dropped
  - The system authority *EXECUTE on the library that contains the global variable to be dropped
  - The DELETE privilege on the SYSVARIABLES catalog table, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To drop an XSR object, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authorities of *OBJOPR and *OBJEXIST on the *SQLXSR object for the XSR object to be dropped
  - The system authority *EXECUTE on the library that contains the XSR object to be dropped
  - The DELETE privilege on the XSROBJECTS, XSROBJECTCOMPONENTS, and XSRANNOTATIONINFO catalog tables, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To drop a function, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - For SQL functions, the system authority *OBJEXIST on the service program object associated with the function, and
  - The DELETE privilege on the SYSFUNCS, SYSPARMS, and SYSROUTINEDEP catalog tables, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To drop a procedure, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - For SQL procedures, the system authority *OBJEXIST on the program object associated with the procedure, and
  - The DELETE privilege on the SYSPROCS, SYSPARMS, and SYSROUTINEDEP catalog tables, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To drop a sequence, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - The system authority *OBJEXIST on the data area associated with the sequence, and
  - The system authority *EXECUTE on the library that contains the sequence to be dropped
  - The DELETE privilege on the SYSSEQOBJECTS catalog table, and
  - The system authority *EXECUTE on library QSYS2, and

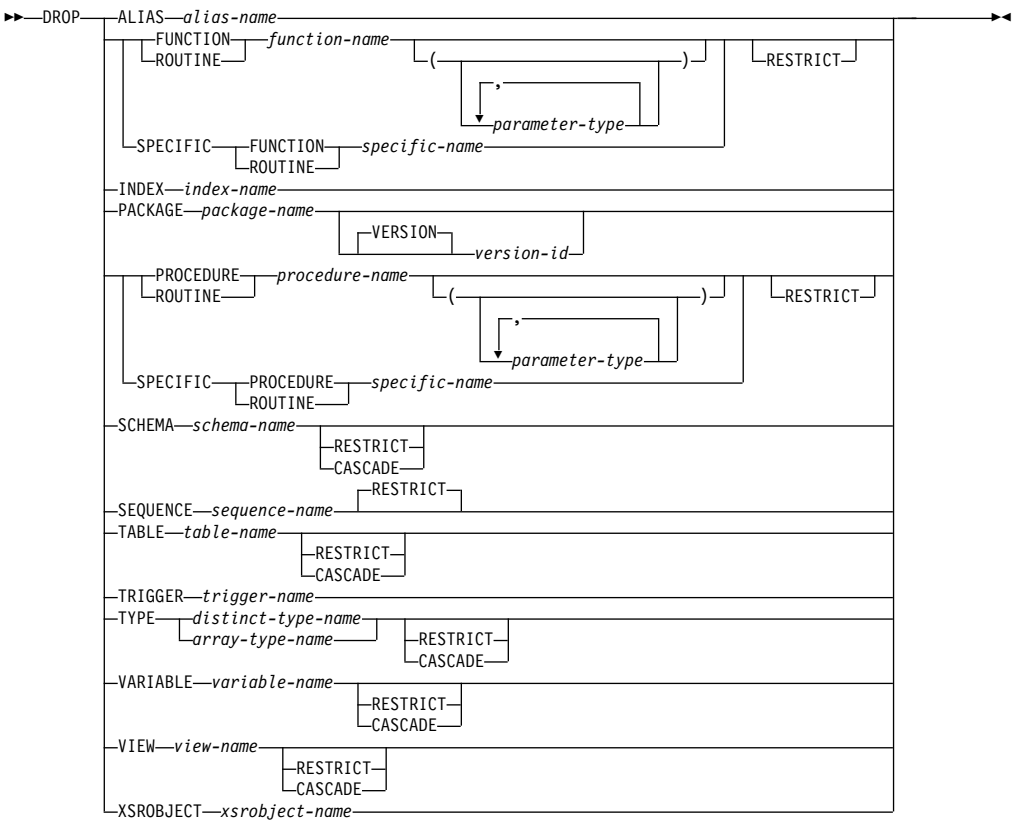
  - *USE to the Delete Data Area (DLTDTAARA) command
- Administrative authority

To drop a trigger, the privileges held by the authorization ID of the statement must include at least one of the following:

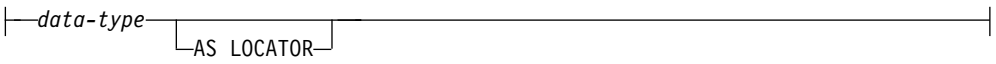
- The following privileges:
  - The system authority *USE to the Remove Physical File Trigger (RMVPFTRG) command, and
  - For the subject table or view of the trigger:
    - The ALTER privilege to the subject table or view, and
    - The system authority *EXECUTE on the library containing the subject table or view,
  - If the trigger being dropped is an SQL trigger:
    - The system authority *OBJEXIST on the trigger program object, and
    - The system authority *EXECUTE on the library containing the trigger.
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

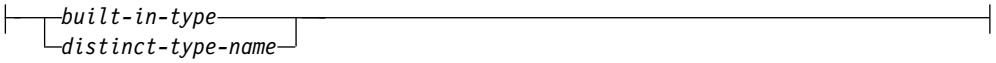
Syntax



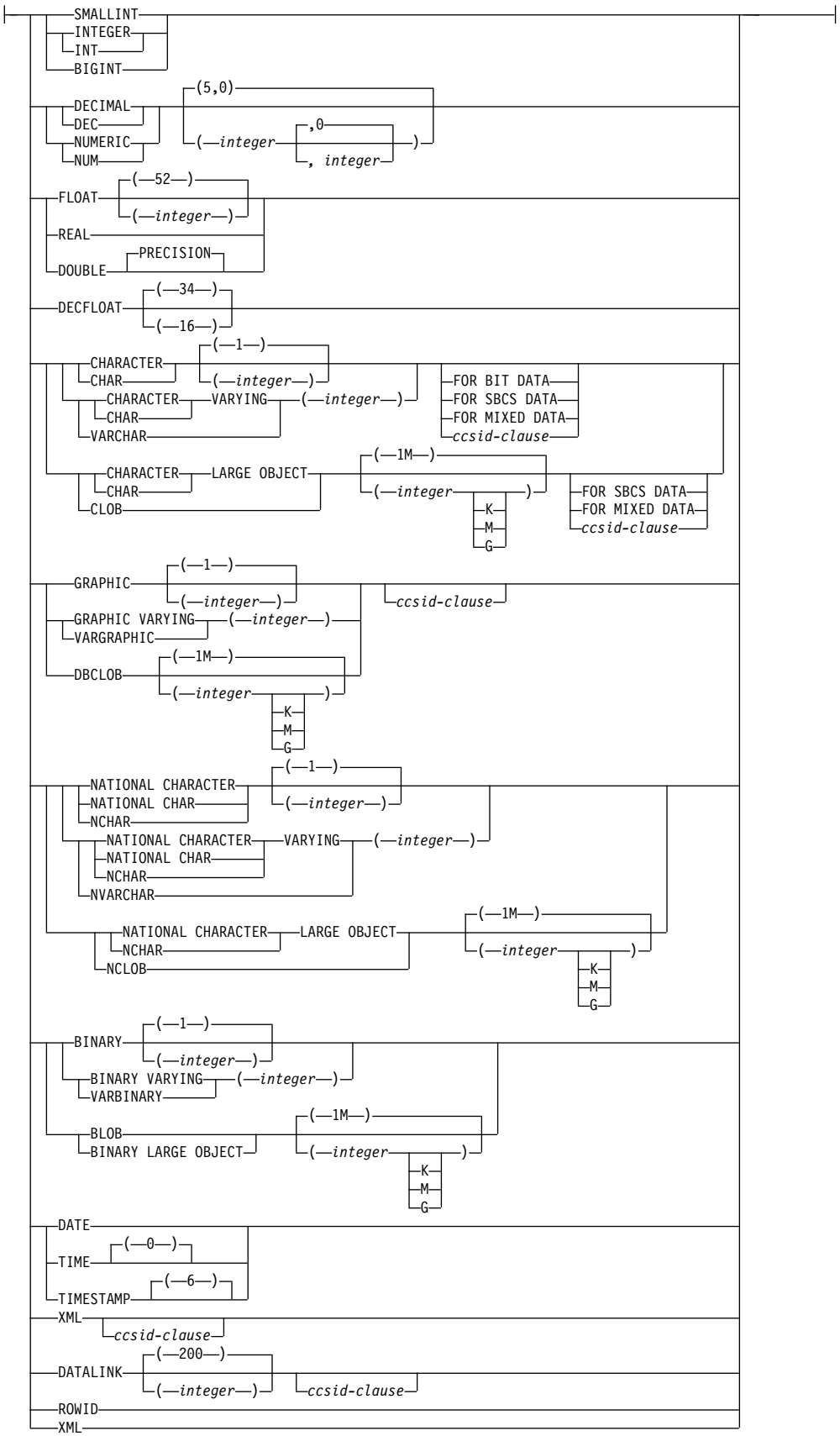
parameter-type:



data-type:



built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(integer
,0
, integer
)
FLOAT
(52)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
FOR BIT DATA
FOR SBCS DATA
FOR MIXED DATA
ccsid-clause
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
K
M
G
GRAPHIC
VARGRAPHIC
GRAPHIC VARYING
DBCLOB
NATIONAL CHARACTER
NATIONAL CHAR
NCHAR
NATIONAL CHARACTER VARYING
NATIONAL CHAR VARYING
NVARCHAR
NATIONAL CHARACTER LARGE OBJECT
NCLOB
BINARY
BINARY VARYING
VARBINARY
BLOB
BINARY LARGE OBJECT
DATE
TIME
TIMESTAMP
(0)
(6)
XML
DATALINK
(200)
ROWID

ccsid-clause:

Description

ALIAS *alias-name*
Identifies the alias that is to be dropped. The *alias-name* must identify an alias that exists at the current server.

The specified alias is deleted from the schema. Dropping an alias has no effect on any constraint, view, or materialized query that was defined using the alias. An alias can be dropped whether it is referenced in a function, package, procedure, program, trigger, or variable.

FUNCTION or SPECIFIC FUNCTION
Identifies the function that is to be dropped. The function must exist at the current server and it must be a function that was defined with the CREATE FUNCTION statement. The particular function can be identified by its name, function signature, or specific name.

Functions implicitly generated by the CREATE TYPE statement cannot be dropped using the DROP statement. They are implicitly dropped when the distinct type is dropped.

The function cannot be dropped if another function is dependent on it. A function is dependent on another function if it was identified in the SOURCE clause of the CREATE FUNCTION statement. A function can be dropped whether it is referenced in a function, package, procedure, program, trigger, variable, or view unless RESTRICT is specified

The specified function is dropped from the schema. All privileges on the user-defined function are also dropped. If this is an SQL function or sourced function, the service program (*SRVPGM) associated with the function is also dropped. If this is an external function, the information that was saved in the program or service program specified on the CREATE FUNCTION statement is removed from the object.

FUNCTION *function-name*
Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*
Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is to be dropped. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name
Identifies the name of the function.

(parameter-type, ...)
Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

RESTRICT
Specifies that the function cannot be dropped if it is referenced in an SQL function, SQL procedure, table, SQL trigger, variable, or view.

INDEX *index-name*
Identifies the index that is to be dropped. The *index-name* must identify an index that exists at the current server.

The specified index is dropped from the schema. An index can be dropped whether it is referenced in a function, package, procedure, program, or trigger.

PACKAGE *package-name*
Identifies the package that is to be dropped. The *package-name* must identify a package that exists at the current server.

The specified package is dropped from the schema. All privileges on the package are also dropped.

A package can be dropped whether it is referenced in a function, package, procedure, program, or trigger.

VERSION *version-id*
version-id is the version identifier that was assigned to the package when it was created. If *version-id* is not specified, a null string is used as the version identifier.

PROCEDURE or SPECIFIC PROCEDURE
Identifies the procedure that is to be dropped. The *procedure-name* must identify a procedure that exists at the current server.

The specified procedure is dropped from the schema. All privileges on the procedure are also dropped. If this is an SQL procedure, the program (*PGM) or service program (*SRVPGM) associated with the procedure is also dropped. If this is an external procedure, the information that was saved in the program specified on the CREATE PROCEDURE statement is removed from the object.

A procedure can be dropped whether it is referenced in a function, package, procedure, program, or trigger.

PROCEDURE *procedure-name*
Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*
Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be dropped. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name
Identifies the name of the procedure.

(parameter-type, ...)
Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If

the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).

- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

RESTRICT
Specifies that the procedure cannot be dropped if it is referenced in an SQL function, SQL procedure, or SQL trigger.

SCHEMA *schema-name*
Identifies the schema that is to be dropped. The *schema-name* must identify a schema that exists at the current server.

The specified schema is dropped. Each object in the schema is dropped as if the appropriate DROP statement was executed with the specified drop option (CASCADE, RESTRICT, or neither). See the DROP description of these object types for information about the handling of objects dependent on these objects.

DROP SCHEMA is only valid when the commit level is *NONE.

Neither CASCADE nor RESTRICT
Specifies that the schema will be dropped even if it is referenced in a function, package, procedure, program, table, trigger, or variable in another schema.

CASCADE
Specifies that any objects in the schema and any triggers that reference the schema will be dropped.

RESTRICT
Specifies that the schema cannot be dropped if it is referenced in an SQL trigger in another schema or if the schema contains any SQL objects other than catalog views, the journal, and journal receiver.

SEQUENCE *sequence-name*
Identifies the sequence that is to be dropped. The *sequence-name* must identify a sequence that exists at the current server.

RESTRICT
Specifies that the sequence cannot be dropped if it is referenced in an SQL trigger, function, procedure, or variable.

TABLE *table-name*
Identifies the table that is to be dropped. The *table-name* must identify a base table that exists at the current server, but must not identify a catalog table.

The specified table is dropped from the schema. All privileges, constraints, indexes, and triggers on the table are also dropped.

Any aliases that reference the specified table are not dropped.

Neither CASCADE nor RESTRICT
Specifies that the table will be dropped even if it is referenced in a constraint, index, trigger, view, or materialized query table. All indexes, views, and materialized query tables that reference the table are dropped even if the authorization ID of the statement does not explicitly have privileges to those objects.

CASCADE
Specifies that the table will be dropped even if it is referenced in a constraint, index, trigger, variable, view, XSR object, or materialized query table. All constraints, indexes, triggers, variables, views, XSR objects and materialized query tables that reference the table are dropped even if the authorization ID of the statement does not explicitly have privileges to those objects.

RESTRICT
Specifies that the table cannot be dropped if it is referenced in a constraint, index, trigger, variable, view, XSR object, or materialized query table.

TRIGGER *trigger-name*
Identifies the trigger that is to be dropped. The *trigger-name* must identify a trigger that exists at the current server.

The specified trigger is dropped from the schema. If the trigger is an SQL trigger, the program object associated with the trigger is also deleted from the schema.

If *trigger-name* specifies an INSTEAD OF trigger on a view, another trigger may depend on that trigger through an update against the view.

TYPE *distinct-type-name* **or** *array-type-name*
Identifies the type that is to be dropped. The *distinct-type-name* or *array-type-name* must identify a distinct type or array type that exists at the current server. The specified type is deleted from the schema.

Neither CASCADE nor RESTRICT
Specifies that the type cannot be dropped if any constraints, indexes, sequences, tables, variables, and views reference the type.

For every procedure or function `R` that has parameters or a return value of the type being dropped, the following DROP statement is effectively executed:

```
DROP ROUTINE R
```

For every trigger `T` that references the type being dropped, the following DROP statement is effectively executed:

```
DROP TRIGGER T
```

It is possible that this statement would cascade to drop dependent functions or procedures. If all of these functions or procedures are in the list to be dropped because of a dependency on the type, the drop of the type will succeed.

CASCADE
Specifies that the type will be dropped even if it is referenced in a constraint, function, index, procedure, sequence, table, trigger, variable, or view. All constraints, functions, indexes, procedures, sequences, tables, triggers, variables, and views that reference the type are dropped. The cascade processing is limited to cases where the type is used to define the routine parameter type, sequence type, variable type and column type.

RESTRICT
Specifies that the type cannot be dropped if it is referenced in a constraint, function (other than a function that was created when the type was created), index, procedure, sequence, table, trigger, variable, or view. The restrict checking is limited to cases where the type is used to define the routine parameter type, sequence type, variable type and column type.

VARIABLE *variable-name*
Identifies the variable that is to be dropped. The *variable-name* must identify a variable that exists at the current server. The specified variable is dropped from the schema.

Neither CASCADE nor RESTRICT
Specifies that the variable will be dropped even if it is referenced in a trigger, procedure, function, view, or another variable. All tables and views that reference the variable are dropped.

CASCADE
Specifies that the variable will be dropped even if it is referenced in a table, trigger, procedure, function, view, or another variable. All tables, triggers, procedures, functions, views, and variables that reference the variable are dropped.

RESTRICT
Specifies that the variable cannot be dropped if it is referenced in a table, trigger, procedure, function, view, or another variable.

VIEW *view-name*
Identifies the view that is to be dropped. The *view-name* must identify a view that exists at the current server, but must not identify a catalog view.

The specified view is dropped from the schema. When a view is dropped, all privileges and triggers on that view are dropped.

Neither CASCADE nor RESTRICT
Specifies that the view will be dropped even if it is referenced in a trigger, materialized query table, or another view. All views and materialized query tables that reference the view are dropped.

CASCADE
Specifies that the view will be dropped even if it is referenced in a trigger, variable, materialized query table, or another view. All triggers, variables, materialized query tables, and views that reference the view are dropped.

RESTRICT
Specifies that the view cannot be dropped if it is referenced in a trigger, variable, materialized query table, or another view.

XSROBJECT *xsrobject-name*
Identifies the XSR object that is to be dropped. The *xsrobject-name* must identify an XSR object that exists at the current server. The specified XSR object is dropped.

Notes

Drop effects: Whenever an object is dropped, its description is dropped from the catalog. If the object is referenced in a function, package, procedure, program, trigger, or variable; any access plans that reference the object are implicitly prepared again when the access plan is next used. If the object does not exist at that time, an error is returned.

Dependencies: Whenever an object is directly or indirectly dropped, other objects that depend on the dropped object might also be dropped. The following semantics determine what happens to a dependent object when the object that it depends on (the underlying object) is dropped:

Three different types of dependencies are shown:

D Dependent object is dropped.

A Automatic revalidation is required. The database manager will attempt to revalidate the object when it is referenced.

R DROP statement fails.

Table 81. Effect of dropping objects that have dependencies

Drop Statement	ALIAS	CONSTRAINT	FUNCTION	INDEX	PROCEDURE	SCHEMA	SEQUENCE	TABLE	TRIGGER	TYPE	VARIABLE	VIEW
DROP ALIAS												
DROP FUNCTION			R[1]		A			A	A		A	A
DROP FUNCTION RESTRICT			R		R			R	R		R	R
DROP INDEX												
DROP PACKAGE												
DROP PROCEDURE			A		A				A			
DROP PROCEDURE RESTRICT			R		R				R			
DROP SCHEMA[2]			A		A			A	A		A	A
DROP SCHEMA CASCADE[3]			A		A			A	D		A	A
DROP SCHEMA RESTRICT[4]			A		A			A	R		A	A
DROP SEQUENCE			A		A				A		A	
DROP SEQUENCE RESTRICT			R		R				R		R	
DROP TABLE		D	A	D	A			D	A		A	D
DROP TABLE CASCADE		D	A	D	A			D	D		D	D
DROP TABLE RESTRICT		R	A	R	A			R	R		R	R
DROP TRIGGER												
DROP TYPE		R	D[5]	R	D		R	R	D		R	R
DROP TYPE CASCADE		D	D	D	D		D	D	D		D	D
DROP TYPE RESTRICT		R	R	R	R		R	R	R		R	R
DROP VARIABLE			A		A			D	A		A	D
DROP VARIABLE CASCADE			D		D			D	D		D	D
DROP VARIABLE RESTRICT			R		R			R	R		R	R
DROP VIEW			A					D	A		A	D
DROP VIEW CASCADE			A		A			D	D		D	D
DROP VIEW RESTRICT			A		A			R	R		R	R
DROP XSROBJ			A		A				A		A	A

Table 81. Effect of dropping objects that have dependencies (continued)

Drop Statement	ALIAS	CONSTRAINT	FUNCTION	INDEX	PROCEDURE	SCHEMA	SEQUENCE	TABLE	TRIGGER	TYPE	VARIABLE	VIEW

Notes:

[1] If other user-defined functions are sourced on the function being dropped, the function cannot be dropped.

[2] DROP SCHEMA will drop all objects in the schema. Any objects in other schemas that reference the schema will not be changed and will require automatic revalidation.

[3] DROP SCHEMA CASCADE will drop all objects in the schema. Triggers in other schemas that reference the schema will be dropped. Other objects in other schemas that reference the schema will not be changed and will require automatic revalidation.

[4] DROP SCHEMA RESTRICT will fail if any object other than SQL catalog views, journals, and journal receivers exist in the schema. Objects other than triggers in other schemas that reference the schema will not be changed and will require automatic revalidation.

[5] Any function or procedure that has a parameter or returns value defined as the type will be dropped.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword SYNONYM can be used as a synonym for ALIAS.
- The keywords DATA TYPE or DISTINCT TYPE can be used as a synonym for TYPE.
- The keyword PROGRAM can be used as a synonym for PACKAGE.
- The keyword COLLECTION can be used as a synonym for SCHEMA.

Examples

Example 1: Drop your table named MY_IN_TRAY. Do not allow the drop if any views or indexes are created over this table.

```
DROP TABLE MY_IN_TRAY RESTRICT
```

Example 2: Drop your view named MA_PROJ.

```
DROP VIEW MA_PROJ
```

Example 3: Drop the package named PERS.PACKA.

```
DROP PACKAGE PERS.PACKA
```

Example 4: Drop the distinct type DOCUMENT, if it is not currently in use:

```
DROP DISTINCT TYPE DOCUMENT RESTRICT
```

Example 5: Assume that you are SMITH and that ATOMIC_WEIGHT is the only function with that name in schema CHEM. Drop ATOMIC_WEIGHT.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT RESTRICT
```

Example 6: Drop the function named CENTER, using the function signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTER (INTEGER, FLOAT) RESTRICT
```

Example 7: Drop CENTER, using the specific name to identify the function instance to be dropped.

```
DROP SPECIFIC FUNCTION JOHNSON.FOCUS97
```

Example 8: Assume that procedure OSMOSIS is in schema BIOLOGY. Drop OSMOSIS.

```
DROP PROCEDURE BIOLOGY.OSMOSIS
```

Example 9: Assume that trigger BONUS exists in the default schema. Drop BONUS.

```
DROP TRIGGER BONUS
```

END DECLARE SECTION

The END DECLARE SECTION statement marks the end of an SQL declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in RPG, Java, or REXX.

Authorization

None required.

Syntax

Description

The END DECLARE SECTION statement can be coded in the application program wherever declarations can appear in accordance with the rules of the host language. It is used to indicate the end of an SQL declare section. An SQL declare section starts with a BEGIN DECLARE SECTION statement. For more information about the BEGIN DECLARE SECTION statement, see"BEGIN DECLARE SECTION" on page 801.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and cannot be nested.

Examples

See "BEGIN DECLARE SECTION" on page 801 for examples using the END DECLARE SECTION statement.

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

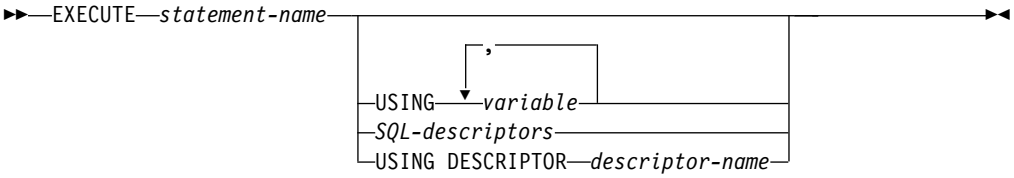
If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

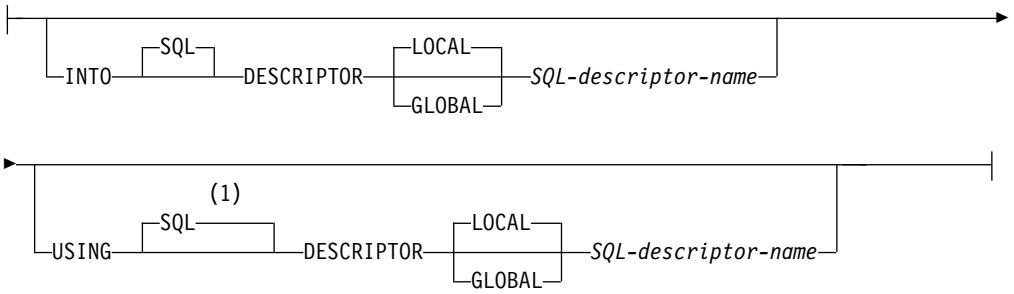
The authorization rules are those defined for the SQL statement specified by EXECUTE. For example, see the description of INSERT for the authorization rules that apply when an INSERT statement is executed using EXECUTE.

The authorization ID of the statement is the run-time authorization ID unless DYNUSRPRF(*OWNER) was specified on the CRTSQLxxx command when the program was created. For more information, see "Authorization IDs and authorization names" on page 68.

Syntax



SQL-descriptors:



Notes:

1 If an SQL descriptor is specified in the USING clause and the INTO clause is not specified, USING DESCRIPTOR is not allowed and USING SQL DESCRIPTOR must be specified.

Description

statement-name
Identifies the prepared statement to be executed. When the EXECUTE statement is executed, the name must identify a prepared statement at the current server. The prepared statement cannot be a SELECT statement.

USING
Introduces a list of variables whose values are substituted for the parameter markers (question marks) in the prepared statement. For an explanation of parameter markers, see "PREPARE" on page 1293. If the prepared statement includes parameter markers, the USING clause must be used. USING is ignored if there are no parameter markers.

variable,...
Identifies one of more host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and variables. A reference to a host structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement.

A global variable may only be used if the current connection is a local connection (not a DRDA connection).

SQL-descriptors

INTO
Identifies an SQL descriptor which contains valid descriptions of the output variables to be used with the EXECUTE statement. This clause is only valid for a CALL or VALUES INTO statement. Before the EXECUTE statement is executed, a descriptor must be allocated using the ALLOCATE DESCRIPTOR statement.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "GET DESCRIPTOR" on page 1182 for an explanation of the information that is placed in the SQL descriptor.

USING
Identifies an SQL descriptor which contains valid descriptions of the input variables to be used with the EXECUTE statement. Before the EXECUTE statement is executed, a descriptor must be allocated using the ALLOCATE DESCRIPTOR statement.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation. The information is returned from the descriptor known in this local scope.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the

SQL session. The information is returned from the descriptor known to any program that executes using the same database connection.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "SET DESCRIPTOR" on page 1361 for an explanation of the information in the SQL descriptor.

DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of variables.

Before the EXECUTE statement is processed, the user must set the following fields in the SQLDA. (The rules for REXX are different. For more information, see the Embedded SQL Programming topic collection.)

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. If LOBs or distinct types are present in the results, there must be additional SQLVAR entries for each parameter. For more information about the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see "Determining how many SQLVAR occurrences are needed" on page 1526.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement. The *n*th variable described by the SQLDA corresponds to the *n*th parameter marker in the prepared statement.

Note that RPG/400 does not provide the function for setting pointers. Because the SQLDA uses pointers to locate the appropriate variables, you have to set these pointers outside your RPG/400 application.

Notes

Parameter marker replacement: Before the prepared statement is executed, each parameter marker in the statement is effectively replaced by its corresponding variable. The replacement of a parameter marker is an assignment operation in which the source is the value of the variable, and the target is a variable within the database manager. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker. For the rules that affect parameter markers, see Table 105 on page 1300.

Let V denote a variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P using storage assignment rules as described in "Assignments and comparisons" on page 98. Thus:

- V must be compatible with the target.

- If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, the value of V must not be null.

However, unlike the storage assignment rules:

- If V is a string, the value will be truncated (without an error), if its length is greater than the length attribute of the target.

When the prepared statement is executed, the value used in place of P is the value of the target variable for P. For example, if V is CHAR(6) and the target is CHAR(8), the value used in place of P is the value of V padded with two blanks.

Example

This example of portions of a COBOL program shows how an INSERT statement with parameter markers is prepared and executed.

```
EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.
  77 EMP              PIC X(6).
  77 PRJ              PIC X(6).
  77 ACT              PIC S9(4) COMP-4.
  77 TIM              PIC S9(3)V9(2).
  01 HOLDER.
     49  HOLDER-LENGTH    PIC S9(4) COMP-4.
     49  HOLDER-VALUE     PIC X(80).
EXEC SQL  END DECLARE SECTION  END-EXEC.
.
.
.
MOVE 70 TO HOLDER-LENGTH.
MOVE "INSERT INTO EMPPROJACT (EMPNO, PROJNO, ACTNO, EMPTIME)
-              "VALUES (?, ?, ?, ?)" TO HOLDER-VALUE.
EXEC SQL  PREPARE MYINSERT FROM :HOLDER  END-EXEC.

IF SQLCODE = 0
  PERFORM DO-INSERT THRU END-DO-INSERT
ELSE
  PERFORM ERROR-CONDITION.

DO-INSERT.
  MOVE "000010" TO EMP.
  MOVE "AD3100" TO PRJ.
  MOVE 160      TO ACT.
  MOVE .50      TO TIM.
  EXEC SQL  EXECUTE MYINSERT USING :EMP, :PRJ, :ACT, :TIM  END-EXEC.
END-DO-INSERT.
.
.
.
```

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements. It can be used to prepare and execute SQL statements that contain neither variables nor parameter markers.

The EXECUTE IMMEDIATE statement:

- Prepares an executable form of an SQL statement from a character string form of the statement
- Executes the SQL statement

Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

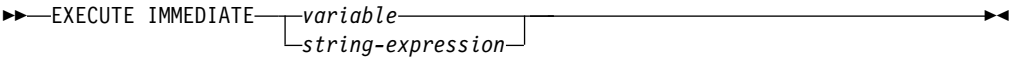
If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

The authorization rules are those defined for the SQL statement specified by EXECUTE IMMEDIATE. For example, see "INSERT" on page 1252 for the authorization rules that apply when an INSERT statement is executed using EXECUTE IMMEDIATE.

The authorization ID of the statement is the run-time authorization ID unless DYNUSRPRF(*OWNER) was specified on the CRTSQLxxx command when the program was created. For more information, see "Authorization IDs and authorization names" on page 68.

Syntax



Description

variable
Identifies a variable that must be declared in accordance with the rules for declaring character-string or Unicode graphic variables. An indicator variable must not be specified.

A global variable may only be used if the current connection is a local connection (not a DRDA connection).

string-expression
A *string-expression* is any PL/I *string-expression* that yields a character string. SQL expressions that yield a character string are not allowed. A *string-expression* is only allowed in PL/I.

The value of the identified variable or string expression is called a *statement string*.

The statement string must be one of the following SQL statements:[100]

ALTER	LABEL	SET CURRENT DECFLOAT ROUNDING MODE
CALL	LOCK TABLE	SET CURRENT DEGREE
COMMENT	MERGE	SET CURRENT IMPLICIT XMLPARSE OPTION
COMMIT	REFRESH TABLE	SET ENCRYPTION PASSWORD
CREATE	RELEASE SAVEPOINT	SET PATH
DECLARE GLOBAL TEMPORARY TABLE	RENAME	SET SCHEMA
DELETE	REVOKE	SET SESSION AUTHORIZATION
DROP	ROLLBACK	SET TRANSACTION
GRANT	SAVEPOINT	SET variable[99]
INSERT	SET CURRENT DEBUG MODE	UPDATE

The statement string must not:

- Begin with EXEC SQL.
- End with END-EXEC or a semicolon.
- Include references to variables.
- Include parameter markers.

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is not valid, it is not executed and the error condition that prevents its execution is reported in the stand-alone SQLSTATE and SQLCODE. If the SQL statement is valid, but an error occurs during its execution, that error condition is reported in the stand-alone SQLSTATE and SQLCODE. Additional information about the error can be retrieved from the SQL Diagnostics Area (or the SQLCA).

Note

Performance considerations: If the same SQL statement is to be executed more than once, it is more efficient to use the PREPARE and EXECUTE statements rather than the EXECUTE IMMEDIATE statement.

Example

Use C to execute the SQL statement in the variable Qstring.

```
void main ()
  {

    EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.

    char Qstring[100] = "INSERT INTO WORK_TABLE SELECT * FROM EMPPROJACT
     WHERE ACTNO >= 100";

    EXEC SQL  END DECLARE SECTION  END-EXEC.
    EXEC SQL  INCLUDE SQLCA;
  .
  .
```

99. The target of the SET variable statement must be a global variable.

100. A *select-statement* is not allowed. To dynamically process a *select-statement*, use the PREPARE, DECLARE CURSOR, and OPEN statements.

```
.
  EXEC SQL EXECUTE IMMEDIATE :Qstring;

  return;
}
```

FETCH

The FETCH statement positions a cursor on a row of the result table. It can return zero, one, or multiple rows, and it assigns the values of the rows returned to variables.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. Multiple row fetch is not allowed in a REXX procedure.

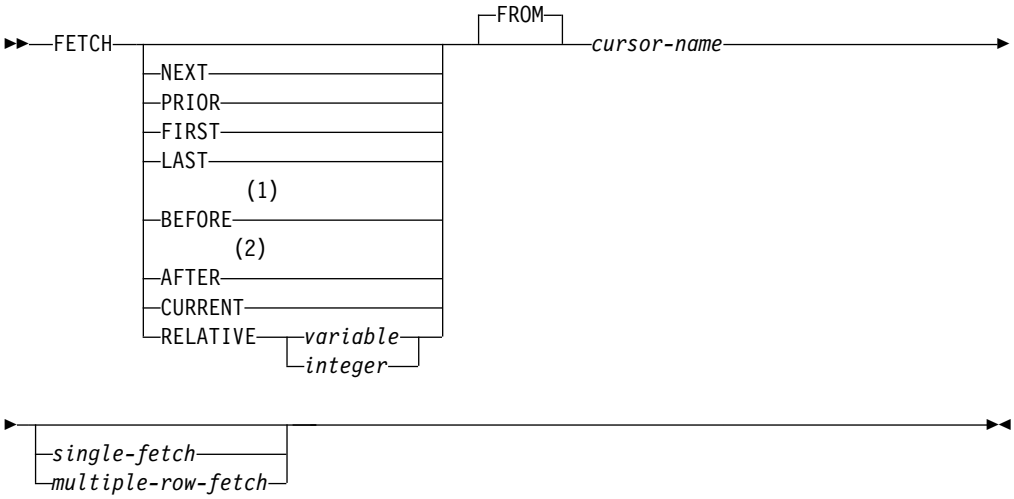
Authorization

See "DECLARE CURSOR" on page 1079 for an explanation of the authorization required to use a cursor.

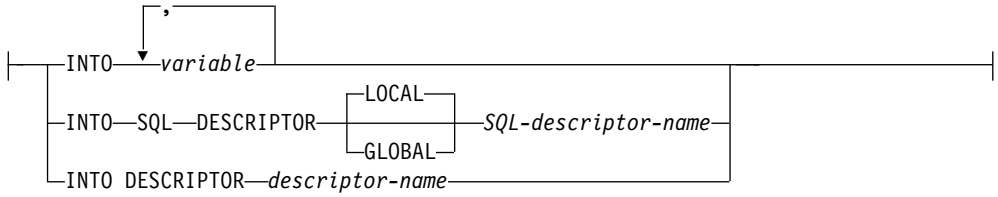
If a global variable is specified in the INTO variable list, the privileges held by the authorization ID of the statement must include:

- The WRITE privilege on the global variable.

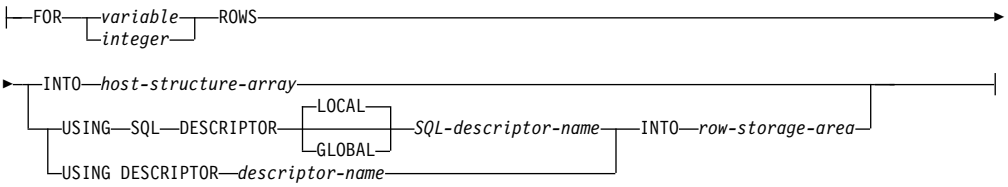
Syntax



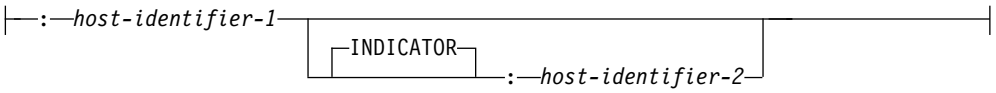
single-fetch:



multiple-row-fetch:



row-storage-area:



Notes:

1 If BEFORE is specified, a *single-fetch* or *multiple-row-fetch* must not be specified.

2 If AFTER is specified, a *single-fetch* or *multiple-row-fetch* must not be specified.

Description

The following keywords specify a new position for the cursor: NEXT, PRIOR, FIRST, LAST, BEFORE, AFTER, CURRENT, and RELATIVE. Of those keywords, only NEXT may be used for cursors that have not been declared SCROLL.

NEXT
Positions the cursor on the next row of the result table relative to the current cursor position. NEXT is the default if no other cursor orientation is specified.

PRIOR
Positions the cursor on the previous row of the result table relative to the current cursor position.

FIRST
Positions the cursor on the first row of the result table.

LAST
Positions the cursor on the last row of the result table.

BEFORE
Positions the cursor before the first row of the result table.

AFTER
Positions the cursor after the last row of the result table.

CURRENT
Does not reposition the cursor, but maintains the current cursor position. If the cursor has been declared as DYNAMIC SCROLL and the current row has been updated so its place within the sort order of the result table is changed, an error is returned.

RELATIVE
Variable or *integer* is assigned to an integer value *k*. RELATIVE positions the cursor to the row in the result table that is either *k* rows after the current row if *k*>0, or *k* rows before the current row if *k*<0. If a *variable* is specified, it must be a numeric variable with zero scale and it must not include an indicator variable.

Table 82. Synonymous Scroll Specifications

Specification	Alternative
RELATIVE +1	NEXT
RELATIVE -1	PRIOR
RELATIVE 0	CURRENT

FROM
This keyword is provided for clarity only. If a scroll position option is specified, then this keyword is required. If no scrolling option is specified, then the FROM keyword is optional.

cursor-name
Identifies the cursor to be used in the fetch operation. The *cursor-name* must identify a declared cursor as explained in "Description" on page 1080 for the DECLARE CURSOR statement or when used in Java, an instance of an SQLJ iterator. When the FETCH statement is executed, the cursor must be in the open state.

If a *single-fetch* or *multiple-row-fetch* clause is not specified, no data is returned to the user. However, the cursor is positioned and a row lock may be acquired. For more information about locking, see "Isolation level" on page 26.

single-fetch

INTO *variable*,...
Identifies one or more host structures or variables that must be declared in accordance with the rules for declaring host structures and variables. In the operational form of INTO, a host structure is replaced by a reference to each of its variables. The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on.

A global variable may only be used if the current connection is a local connection (not a DRDA connection).

INTO SQL DESCRIPTOR *SQL-descriptor-name*
Identifies an SQL descriptor which contains valid descriptions of the output variables to be used with the FETCH statement. Before the FETCH statement is executed, a descriptor must be allocated using the ALLOCATE DESCRIPTOR statement.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "SET DESCRIPTOR" on page 1361 for an explanation of the information in the SQL descriptor.

INTO DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more variables.

Before the FETCH statement is processed, the user must set the following fields in the SQLDA. (The rules for REXX are different. For more information see the Embedded SQL Programming topic collection.)

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to 16 + SQLN*(80), where 80 is the length of an SQLVAR occurrence. If LOBs are specified, there must be two SQLVAR entries for each parameter marker and SQLN must be set to two times the number of parameter markers.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. For more information, see Appendix D, "SQLDA (SQL descriptor area)," on page 1523.

The USING DESCRIPTOR clause is not supported for a FETCH statement within a Java program.

multiple-row-fetch

FOR *variable* **or** *integer* **ROWS**

Evaluates *variable* or *integer* to an integral value that represents the number of rows to fetch. If a *variable* is specified, it must be a numeric variable with zero scale and it must not include an indicator variable. It must not be a global variable. The value must be in the range of 1 to 32767 and the total size of the rows, excluding LOBs, must be less than 16M. The cursor is positioned on the row specified by the orientation keyword (for example, NEXT), and that row is fetched. Then the next rows are fetched (moving forward in the table), until either the specified number of rows have been fetched or the end of the cursor is reached. After the fetch operation, the cursor is positioned on the last row fetched.

For example, FETCH PRIOR FROM C1 FOR 3 ROWS causes the previous row, the current row, and the next row to be returned, in that order. The cursor is positioned on the next row. FETCH RELATIVE -1 FROM C1 FOR 3 ROWS returns the same result. FETCH FIRST FROM C1 FOR :x ROWS returns the first *x* rows, and leaves the cursor positioned on row number *x*.

When a *multiple-row-fetch* is successfully executed, three statement information items are available in the SQL Diagnostics Area (or the SQLCA):

- ROW_COUNT (or SQLERRD(3) of the SQLCA) shows the number of rows retrieved.
- DB2_ROW_LENGTH (or SQLERRD(4) of the SQLCA) contains the length of the row retrieved.
- DB2_LAST_ROW (or SQLERRD(5) of the SQLCA) contains +100 if the last row was fetched. [101]

101. If the number of rows returned is equal to the number of rows requested, then an end of data warning may not occur and DB2_LAST_ROW (or SQLERRD(5) of the SQLCA) may not contain +100.

INTO *host-structure-array*
host-structure-array identifies an array of host structures defined in accordance with the rules for declaring host structures.

The first structure in the array corresponds to the first row, the second structure in the array corresponds to the second row, and so on. In addition, the first value in the row corresponds to the first item in the structure, the second value in the row corresponds to the second item in the structure, and so on. The number of rows to be fetched must be less than or equal to the dimension of the host structure array.

USING SQL DESCRIPTOR *SQL-descriptor-name*
Identifies an SQL descriptor.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

The COUNT field in the descriptor header must be set to reflect the number of columns in the result set. The TYPE and DATETIME_INTERVAL_CODE (if applicable) must be set for each column in the result set.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more variables that describe the format of a row in the *row-storage-area*.

Before the FETCH statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of the variables.

The values of the other fields of the SQLDA (such as SQLNAME) may not be defined after the FETCH statement is executed and should not be used.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to 16 + SQLN*(80), where 80 is the length of an SQLVAR occurrence. If LOBs or distinct types are specified, there must be two SQLVAR entries for each parameter marker and SQLN must be set to two times the number of parameter markers.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. For more information, see Appendix D, “SQLDA (SQL descriptor area),” on page 1523.

On completion of the FETCH, the SQLDATA pointer in the first SQLVAR entry addresses the returned value for the first column in the allocated storage in the

first row, the SQLDATA pointer in the second SQLVAR entry addresses the returned value for the second column in the allocated storage in the first row, and so on. The SQLIND pointer in the first nullable SQLVAR entry addresses the first indicator value, the SQLIND pointer in the second nullable SQLVAR entry addresses the second indicator value, and so on. The SQLDA must be allocated on a 16-byte boundary.

INTO *row-storage-area*

host-identifier-1 specified with a variable identifies an allocation of storage in which to return the rows. The rows are returned into the storage area in the format described by the SQLDA or SQL descriptor. *host-identifier-1* must be large enough to hold all the rows requested.

host-identifier-2 identifies the optional indicator area. It should be specified if any of the data types returned are nullable. The indicators are returned as small integers. *host-identifier-2* must be large enough to contain an indicator for each nullable value for each row to be returned.

The GET DIAGNOSTICS statement can be used to return the DB2_ROW_LENGTH which indicates the length of each row returned into the *row-storage-area*.

The *nth* variable identified by the INTO clause or described in the SQLDA corresponds to the *nth* column of the result table of the cursor. The data type of each variable must be compatible with its corresponding column.

Each assignment to a variable is made according to the retrieval assignment rules described in "Retrieval assignment" on page 102.[102] If the number of variables is less than the number of values in the row, the SQLSTATE is set to '01503' (or the SQLWARN3 field of the SQLCA is set to 'W'). Note that there is no warning if there are more variables than the number of result columns. If the value is null, an indicator variable must be provided. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

If an error occurs as the result of an arithmetic expression in the SELECT list of an outer SELECT statement (division by zero, overflow, etc.) or a character conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided. The value of the variable is undefined. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, a warning is returned.) If you do not provide an indicator variable, an error is returned. It is possible that some values have already been assigned to variables and will remain assigned when the error occurs.

multiple-row-fetch is not allowed if any of the result columns are LOBs or if the current connection is to a remote server.

Notes

Cursor position: An open cursor has three possible positions:

- Before a row
- On a row

102. If assigning to an SQL-variable or SQL-parameter and the standards option is specified, storage assignment rules apply. For information on the standards option, see "Standards compliance" on page xi.

- After the last row

If a cursor is positioned on a row, that row is called the current row of the cursor. A cursor referenced in an UPDATE or DELETE statement must be positioned on a row. A cursor can only be positioned on a row as a result of a FETCH statement.

It is possible for an error to occur that makes the state of the cursor unpredictable.

Variable assignment: The *n*th variable identified by the INTO clause or described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of each variable must be compatible with its corresponding column.

Each assignment to a variable is made according to the Retrieval Assignment rules described in "Assignments and comparisons" on page 98. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLCA is set to 'W'. Note that there is no warning if there are more variables than the number of result columns. If the value is null, an indicator variable must be provided. If an assignment error occurs, the values in the variables are unpredictable.

If the specified variable is a string and is not large enough to contain the result, a warning (SQLSTATE 01004) is returned (and 'W' is assigned to SQLWARN1 in the SQLCA). The actual length of the result is returned in the indicator variable associated with the *variable*, if an indicator variable is provided.

If the specified variable is a C NUL-terminated variable and is not large enough to contain the result and the NUL-terminator:

- If the *CNULRQD option is specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*YES) on the SET OPTION statement), the following occurs:
  - The result is truncated.
  - The last character is the NUL-terminator.
  - A warning (SQLSTATE 01004) is returned (and 'W' is assigned to SQLWARN1 in the SQLCA).
- If the *NOCNULRQD option on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement) is specified, the following occurs:
  - The NUL-terminator is not returned.
  - A warning (SQLSTATE 01004) is returned (and 'N' is assigned to SQLWARN1 in the SQLCA).

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- USING DESCRIPTOR may be used as a synonym for INTO DESCRIPTOR in the single-fetch-clause.

Example

Example 1: In this C example, the FETCH statement fetches the results of the SELECT statement into the program variables `dnum`, `dname`, and `mnum`. When no more rows remain to be fetched, the not found condition is returned.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM TDEPT
  WHERE ADMRDEPT = 'A00';
EXEC SQL OPEN C1;
while (SQLCODE==0) {
  EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
}
EXEC SQL CLOSE C1;
```

Example 2: This FETCH statement uses an SQLDA.

```
FETCH CURS USING DESCRIPTOR :sqlda3
```

FREE LOCATOR

The FREE LOCATOR statement removes the association between a locator variable and its value.

Invocation

This statement can only be embedded in an application program. It cannot be issued interactively. It is an executable statement that can be dynamically prepared. However, the EXECUTE statement with the USING clause must be used to execute the prepared statement. FREE LOCATOR cannot be used with the EXECUTE IMMEDIATE statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax

Description

variable,...
Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables. The locator variable type must be a binary large object locator, a character large object locator, a double-byte character large object locator, or an XML locator.

The *variable* must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a CALL, FETCH, SELECT INTO, assignment statement, SET variable, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is returned.

If more than one locator variable is specified and an error occurs on one of the locators, no locators will be freed.

Example

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that locators have been established in a program to represent the column values. In a COBOL program, free the CLOB locator variables LOCRES and LOCHIST, and the BLOB locator variable LOCPIC.

```
EXEC SQL
FREE LOCATOR :LOCRES, :LOCHIST, :LOCPIC
END-EXEC.
```

GET DESCRIPTOR

The GET DESCRIPTOR statement gets information from an SQL descriptor.

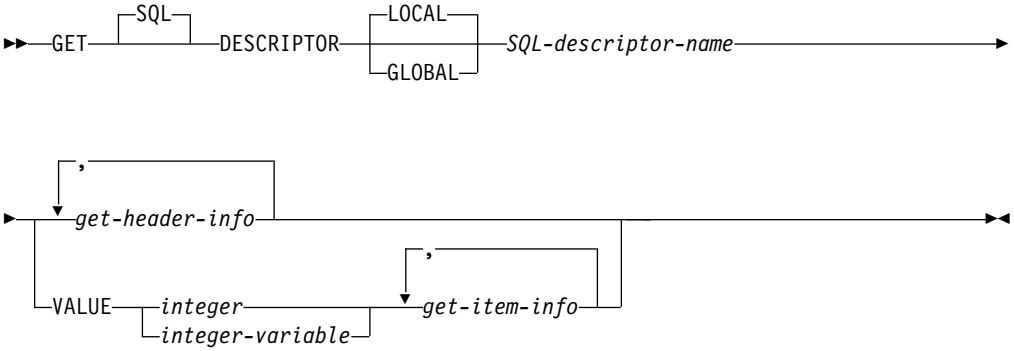
Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It cannot be issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

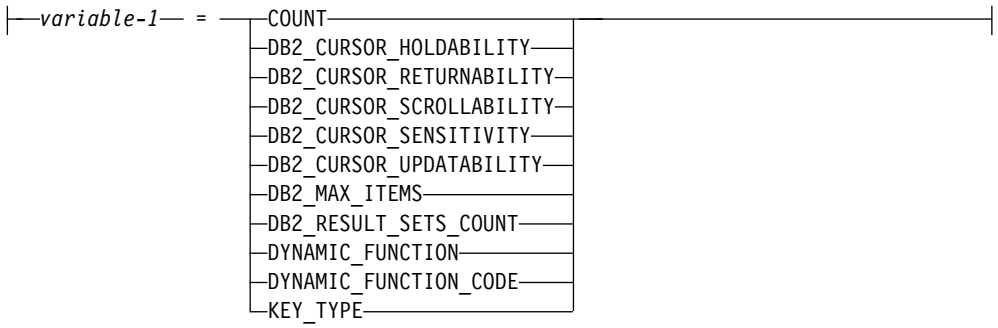
Authorization

None required.

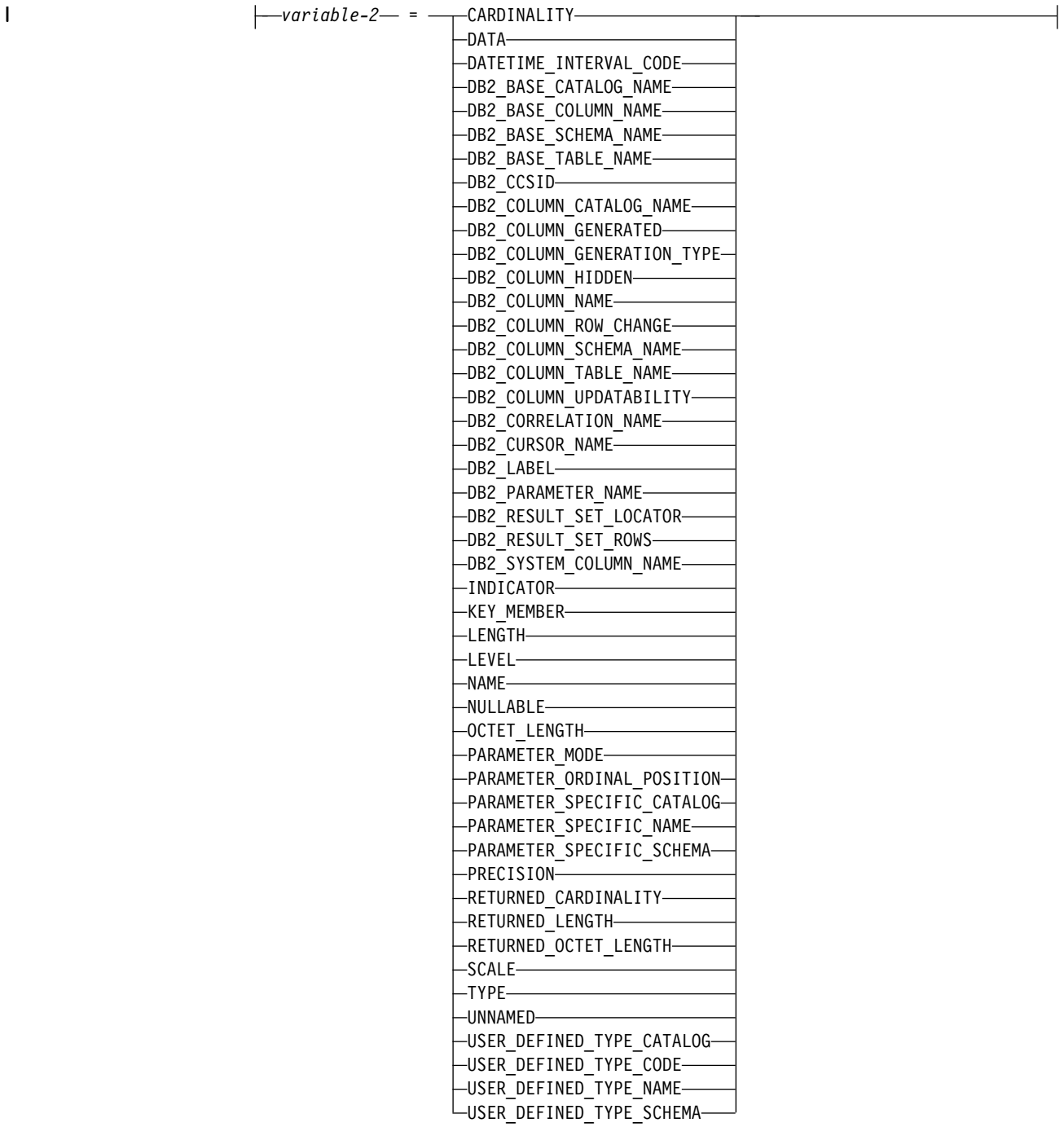
Syntax



get-header-info:



get-item-info:



Description

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation. The information is returned from the descriptor known in this local scope.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session. The information is returned from the descriptor known to any program that executes using the same database connection.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

get-header-info
Returns information about the prepared SQL statement and SQL descriptor.

VALUE
Identifies the item number for which the specified information is retrieved. If the value is greater than the value of COUNT (from the header information), then no result is returned. If the item number is greater than the maximum number of items allocated for the descriptor or the item number is less than 1, an error is returned.

integer
An integer constant in the range of 1 to the number of items in the SQL descriptor.

integer-variable
Identifies a variable declared in the program in accordance with the rules for declaring variables. It must not be a global variable. The data type of the variable must be SMALLINT, INTEGER, BIGINT, or DECIMAL or NUMERIC with a scale of zero. The value of *integer-variable* must be in the range of 1 to the maximum number of items in the SQL descriptor.

get-item-info
Returns information about a specific item in the SQL descriptor.

get-header-info

variable-1
Identifies a variable declared in the program in accordance with the rules for declaring variables, but must not be a file reference variable or a global variable. The data type of the variable must be compatible with the descriptor information item as specified in Table 83 on page 1190. The variable is assigned (using storage assignment rules) to the corresponding descriptor item. For details on the assignment rules, see "Assignments and comparisons" on page 98.

COUNT
A count of the number of items in the descriptor.

DB2_CURSOR_HOLDABILITY
The hold status of the cursor. The possible values are:
0 The descriptor is not describing a cursor or the cursor was not declared WITH HOLD.
1 The cursor was declared WITH HOLD.

DB2_CURSOR_RETURNABILITY
The return status of the cursor's result set. The possible values are:
0 The descriptor is not describing a cursor or the cursor's result set is not returnable.
1 The cursor's result set is returnable to the caller of the procedure.
2 The cursor's result set is returnable to the client.

DB2_CURSOR_SCROLLABILITY
The scroll status of the cursor. The possible values are:
0 The descriptor is not describing a cursor or the cursor was not declared with SCROLL.
1 The cursor was declared with SCROLL.

DB2_CURSOR_SENSITIVITY
The sensitivity of the cursor. The possible values are:
0 The descriptor is not describing a cursor.
1 The cursor is SENSITIVE DYNAMIC.
3 The cursor is INSENSITIVE.
4 The cursor is ASENSITIVE.

DB2_CURSOR_UPDATABILITY
Specifies whether the cursor can be used in an UPDATE statement with WHERE CURRENT OF *cursor-name*. The possible values are:
0 The descriptor is not describing a cursor or the cursor cannot be used in an UPDATE statement with WHERE CURRENT OF *cursor-name*.
1 The cursor can be used in an UPDATE statement with WHERE CURRENT OF *cursor-name*.

DB2_MAX_ITEMS
Represents the value specified as the allocated maximum number of item descriptors on the ALLOCATE DESCRIPTOR statement. If the WITH MAX clause was not specified, the value is the default number of maximum items for the ALLOCATE DESCRIPTOR statement.

DB2_RESULT_SETS_COUNT
The number of result sets returned by the procedure. The value will be 0 if this descriptor is not describing a procedure.

DYNAMIC_FUNCTION
The type of the prepared SQL statement as a character string. For information on statement type, see Table 86 on page 1214.

DYNAMIC_FUNCTION_CODE
The statement code representing the type of the prepared SQL statement. For information on statement codes, see Table 86 on page 1214.

KEY_TYPE
The type of key included in the select list. The possible values are:
0 The descriptor is not describing the columns of a query or there are no key columns referenced in the query, or there is no unique key.
1 The select list includes all the columns of the primary key of the base table referenced by the query.
2 The table referenced by the query does not have a primary key but the select list includes a set of columns that are defined as the preferred candidate key. If there is more than one such preferred candidate key included in the select list, the left-most preferred candidate key is used.

get-item-info

variable–2
Identifies a variable declared in the program in accordance with the rules for declaring variables, but must not be a file reference variable or a global variable. The data type of the variable must be compatible with the descriptor information item as specified in Table 83 on page 1190. The variable is assigned (using storage assignment rules) to the corresponding descriptor item. For details on the assignment rules, see "Assignments and comparisons" on page 98.

When getting the DATA item, in general the variable must have the same data type, length, precision, scale, and CCSID as specified in Table 83 on page 1190. For variable-length types, the variable length must not be less than the LENGTH in the descriptor. For C nul-terminated types, the variable length must be at least one greater than the LENGTH in the descriptor.

CARDINALITY
The cardinality of the array data type. If this descriptor is the result of a DESCRIBE, this is the maximum cardinality of the array data type. The cardinality is 0 for all other data types.

DATA
The value for the data described by the item descriptor. If the value of INDICATOR is negative, then the value of DATA is undefined and the INDICATOR *get-item-info* must also be specified in the same statement.

DATETIME_INTERVAL_CODE
Codes that define the specific datetime data type.
0 Descriptor item does not have TYPE value of 9.
1 DATE
2 TIME
3 TIMESTAMP

DB2_BASE_CATALOG_NAME
The server name of the base table for the column represented by the item descriptor.

DB2_BASE_COLUMN_NAME
The name of the column as defined in the base table referenced in the described query, possibly indirectly through a view. If a column name cannot be defined or is not applicable, this item will contain the empty string. The name is returned as case sensitive and without delimiters.

DB2_BASE_SCHEMA_NAME
The schema name of the base table for the column represented by the item descriptor. If a schema name cannot be defined or is not applicable, this item will contain the empty string. The name is returned as case sensitive and without delimiters.

DB2_BASE_TABLE_NAME
The table name of the underlying base table for the column represented by the item descriptor. If a table name cannot be defined or is not applicable, this item will contain the empty string. The name is returned as case sensitive and without delimiters.

DB2_CCSID
The CCSID of character, graphic, or XML data. Value is zero for all types that are not based on character or graphic string or XML types. Value is 65535 for binary types or character types with the FOR BIT DATA attribute.

DB2_COLUMN_CATALOG_NAME
The server name of the referenced table or view for the column represented by the item descriptor. If a column catalog name cannot be defined or is not applicable, this item will contain the empty string.

DB2_COLUMN_GENERATED
Indicates whether a column is generated. Possible values are:
0 Not generated
1 GENERATED ALWAYS
2 GENERATED BY DEFAULT

DB2_COLUMN_GENERATION_TYPE
Indicates how the column is generated. Possible values are:
0 Not generated
1 IDENTITY column
2 ROWID column
4 Row change timestamp column

DB2_COLUMN_HIDDEN
Indicates whether the column represented by the item descriptor is hidden. Possible values are:
0 Not hidden
1 Implicitly hidden
3 Implicitly hidden for optimistic locking

DB2_COLUMN_NAME
The name of the column as defined in the table or view referenced in the described query. If a column name cannot be defined or is not applicable, this item will contain the empty string. The name is returned as case sensitive and without delimiters.

DB2_COLUMN_ROW_CHANGE
Indicates whether the column represented by the item descriptor was added as a result of using the WITH ROW CHANGE COLUMNS prepare attribute. Possible values are:
-1 ROW CHANGE TOKEN (distinct)
-2 ROW CHANGE TOKEN (not distinct)
-3 RID (only valid from a remote relational database)
-4 RID_BIT (only valid from a remote relational database)

DB2_COLUMN_SCHEMA_NAME
The schema name of the referenced table or view for the column represented by the item descriptor. If a column schema name cannot be defined or is not applicable, this item will contain the empty string. The name is returned as case sensitive and without delimiters.

DB2_COLUMN_TABLE_NAME
The table or view name of the referenced table or view for the column represented by the item descriptor. If a column table name cannot be defined or is not applicable, this item will contain the empty string. The name is returned as case sensitive and without delimiters.

DB2_COLUMN_UPDATABILITY
Indicates whether the column represented by the item descriptor is updatable. Possible values are:
0 Not updatable
1 Updatable

DB2_CORRELATION_NAME
The empty string is always returned.

DB2_CURSOR_NAME
The name of the cursor in the procedure for this result set. This is only set when the descriptor is describing a procedure.

DB2_LABEL
The label defined for the column. If there is no label for the column, this item will contain the empty string.

DB2_PARAMETER_NAME
The name of the parameter for the stored procedure. Only returned for a CALL statement. The name is returned as case sensitive and without delimiters.

DB2_RESULT_SET_LOCATOR
The result set locator for this result set. This is only set when the descriptor is describing a procedure.

DB2_RESULT_SET_ROWS
The estimated number of rows in the result set. It is set to the value -1 if the number is unknown. This is only set when the descriptor is describing a procedure.

DB2_SYSTEM_COLUMN_NAME
The system name of the column. If a system name cannot be defined or is not applicable, this item will contain blanks.

INDICATOR
The value for the indicator. A non-negative value is used when the value returned in this descriptor item is given in the DATA field. When extended indicator variables are not enabled, a negative value is used when the value returned in this descriptor item is the null value. When extended indicators are enabled:

- -1, -2, -3, -4, or -6 indicates the value returned in this descriptor is the null value.
- -5 indicates the value returned in this descriptor item is DEFAULT.
- -7 indicates the value returned in this descriptor item is UNASSIGNED.

KEY_MEMBER
An indication of whether this column is part of a key.
0 This column is not part of a key.
1 This column is part of a unique key.
2 This column by itself is a unique key.

LENGTH
Returns the maximum length of the data. If the data type is a character or graphic string, an XML type, or a datetime type, the length represents the number of characters (not bytes). If the data type is a binary string or any other type, the length represents the number of bytes. For a description of data type codes and lengths, see Table 84 on page 1192.

LEVEL
The level of the item descriptor. The value is 0.

NAME
The name associated with the select list column described by the item descriptor. The name is returned as case sensitive and without delimiters.

NULLABLE
Indicates whether the column or parameter marker is nullable.
0 The select list column or parameter marker cannot have a null value.
1 The select list column or parameter marker can have a null value.

OCTET_LENGTH
Returns the maximum length of the data in bytes for all types. For a description of data type codes and lengths, see Table 84 on page 1192.

PARAMETER_MODE
The mode of the parameter marker in a CALL statement.
0 The descriptor is not associated with a CALL statement.
1 Input only parameter.
2 Input and output parameter.
4 Output only parameter.

PARAMETER_ORDINAL_POSITION
The ordinal position of the parameter marker in a CALL statement. The value is 0 if the descriptor is not associated with a CALL statement.

PARAMETER_SPECIFIC_CATALOG
The server name of the procedure containing the parameter marker.

PARAMETER_SPECIFIC_NAME
The specific name of the procedure containing the parameter marker. The name is returned as case sensitive and without delimiters.

PARAMETER_SPECIFIC_SCHEMA
The schema name of the procedure containing the parameter marker. The name is returned as case sensitive and without delimiters.

PRECISION
Returns the precision for the data:
SMALLINT
5
INTEGER
10
BIGINT
19
NUMERIC and DECIMAL
Defined precision
REAL 24
DOUBLE
53
DECFLOAT(7)
7
DECFLOAT(16)
16
DECFLOAT(34)
34
TIME 0
TIMESTAMP
6
Other data types
0

RETURNED_CARDINALITY
The current cardinality for an array data type returned by FETCH or CALL. The value is 0 when the data type of the item is not an array.

RETURNED_LENGTH
The returned length in characters for character string, graphic string, and XML data types. The returned length in bytes for binary string data types.

RETURNED_OCTET_LENGTH
The returned length in bytes for all string data types.

SCALE
Returns the defined scale if the data type is DECIMAL or NUMERIC. The scale is 0 for all other data types.

TYPE
Returns a data type code representing the data type of the item. For a description of the data type codes and lengths, see Table 84 on page 1192.

UNNAMED
A value of 1 indicates that the NAME value is generated by the database manager. Otherwise, the value is zero and NAME is the derived name of the column in the select list.

USER_DEFINED_TYPE_CATALOG
The server name of the user-defined type. If the type is not a user-defined data type, this item contains the empty string.

USER_DEFINED_TYPE_CODE
Indicates whether the type of the descriptor item is a user-defined type.
0 The descriptor item is not a user-defined type.
1 The descriptor item is a user-defined type.

USER_DEFINED_TYPE_NAME
The name of the user-defined data type. If the type is not a user-defined data type, this item contains the empty string. The name is returned as case sensitive and without delimiters.

USER_DEFINED_TYPE_SCHEMA
The schema name of the user-defined data type. If the type is not a user-defined data type, this item contains the empty string. The name is returned as case sensitive and without delimiters.

Notes

Data types for items: The following table shows the SQL data type for each descriptor item. When a descriptor item is assigned to a variable, the variable must be compatible with the data type of the descriptor item.

Table 83. Data Types for GET DESCRIPTOR Items

Item Name	Data Type
Header Information	
COUNT	INTEGER
DB2_CURSOR_HOLDABILITY	INTEGER
DB2_CURSOR_RETURNABILITY	INTEGER
DB2_CURSOR_SCROLLABILITY	INTEGER
DB2_CURSOR_SENSITIVITY	INTEGER
DB2_CURSOR_UPDATABILITY	INTEGER
DB2_MAX_ITEMS	INTEGER
DB2_RESULT_SETS_COUNT	INTEGER
DYNAMIC_FUNCTION	VARCHAR(128)
DYNAMIC_FUNCTION_CODE	INTEGER
KEY_TYPE	INTEGER
Item Information	
CARDINALITY	BIGINT
DATA	Matches the data type specified by TYPE
DATETIME_INTERVAL_CODE	INTEGER
DB2_BASE_CATALOG_NAME	VARCHAR(128)
DB2_BASE_COLUMN_NAME	VARCHAR(128)
DB2_BASE_SCHEMA_NAME	VARCHAR(128)
DB2_BASE_TABLE_NAME	VARCHAR(128)
DB2_CCSID	INTEGER
DB2_COLUMN_CATALOG_NAME	VARCHAR(128)

Table 83. Data Types for GET DESCRIPTOR Items (continued)

Item Name	Data Type
DB2_COLUMN_GENERATED	INTEGER
DB2_COLUMN_GENERATION_TYPE	INTEGER
DB2_COLUMN_HIDDEN	INTEGER
DB2_COLUMN_NAME	VARCHAR(128)
DB2_COLUMN_ROW_CHANGE	INTEGER
DB2_COLUMN_SCHEMA_NAME	VARCHAR(128)
DB2_COLUMN_TABLE_NAME	VARCHAR(128)
DB2_COLUMN_UPDATABILITY	INTEGER
DB2_CORRELATION_NAME	VARCHAR(128)
DB2_CURSOR_NAME	VARCHAR(128)
DB2_LABEL	VARCHAR(60)
DB2_PARAMETER_NAME	VARCHAR(128)
DB2_RESULT_SET_LOCATOR	Result Set Locator
DB2_RESULT_SET_ROWS	BIGINT
DB2_SYSTEM_COLUMN_NAME	CHAR(10)
INDICATOR	INTEGER
KEY_MEMBER	INTEGER
LENGTH	INTEGER
LEVEL	INTEGER
NAME	VARCHAR(128)
NULLABLE	INTEGER
OCTET_LENGTH	INTEGER
PARAMETER_MODE	INTEGER
PARAMETER_ORDINAL_POSITION	INTEGER
PARAMETER_SPECIFIC_CATALOG	VARCHAR(128)
PARAMETER_SPECIFIC_NAME	VARCHAR(128)
PARAMETER_SPECIFIC_SCHEMA	VARCHAR(128)
PRECISION	INTEGER
RETURNED_CARDINALITY	INTEGER
RETURNED_LENGTH	INTEGER
RETURNED_OCTET_LENGTH	INTEGER
SCALE	INTEGER
TYPE	INTEGER
UNNAMED	INTEGER
USER_DEFINED_TYPE_CATALOG	VARCHAR(128)
USER_DEFINED_TYPE_NAME	VARCHAR(128)
USER_DEFINED_TYPE_SCHEMA	VARCHAR(128)
USER_DEFINED_TYPE_CODE	VARCHAR(128)

SQL data type codes and lengths: The following table represents the possible values for TYPE, LENGTH, OCTET_LENGTH, and DATETIME_INTERVAL_CODE descriptor items.

The values in the following table are assigned by the ISO and ANSI SQL Standard and may change as the standard evolves. Include *sqlscds* in the include source files in library QSYSINC should be used when referencing these values.

Table 84. SQL Data Type Codes and Lengths

Data Type	Data Type Code	Length	Octet Length
SMALLINT	5	2	2
INTEGER	4	4	4
BIGINT	25	8	8
DECIMAL	3	(precision/2)+1	(precision/2)+1
NUMERIC(n)	2	n	n
REAL	7	4	4
FLOAT	6	8	8
DOUBLE PRECISION	8	8	8
DECFLOAT(7)	-360	4	4
DECFLOAT(16)	-360	8	8
DECFLOAT(34)	-360	16	16
CHARACTER(n)	1	n	n
VARCHAR(n)	12	<=n	n
CLOB(n)	40	<=n	n
GRAPHIC(n)	–95	n	2*n
VARGRAPHIC(n)	–96	<=n	2*n
DBCLOB(n)	-350	<=n	2*n
BINARY(n)	–2	n	n
VARBINARY(n)	–3	<=n	n
BLOB(n)	30	n	n
DATE (DATETIME_INTERVAL_CODE = 1)	9	Length depends on date format	Based on CCSID
TIME (DATETIME_INTERVAL_CODE = 2)	9	Length depends on time format	Based on CCSID
TIMESTAMP (DATETIME_INTERVAL_CODE = 3)	9	26	26 or 52 (based on CCSID)
DATALINK(n)	70	<=n	n
ROWID	–904	40	40
XML	137	0	0
C nul terminated CHARACTER(n)	1	<=n	n
C nul terminated GRAPHIC(n)	–400	<=n	2*n
BLOB File Reference Variable	–916	267	267
CLOB File Reference Variable	–920	267	267

Table 84. SQL Data Type Codes and Lengths (continued)

Data Type	Data Type Code	Length	Octet Length
DBCLOB File Reference Variable	–924	267	267
Result Set Locator	-972	8	8
Array	50	N/A	N/A

Example

Example 1: Retrieve from the descriptor 'NEWDA' the number of descriptor items.

```
EXEC SQL GET DESCRIPTOR  'NEWDA'
  :numitems = COUNT;
```

Example 2: Retrieve from the first item descriptor of descriptor 'NEWDA' the data type and the octet length.

```
GET DESCRIPTOR 'NEWDA'
  VALUE 1 :dtype   = TYPE,
          :olength = OCTET_LENGTH;
```

GET DIAGNOSTICS

The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed.

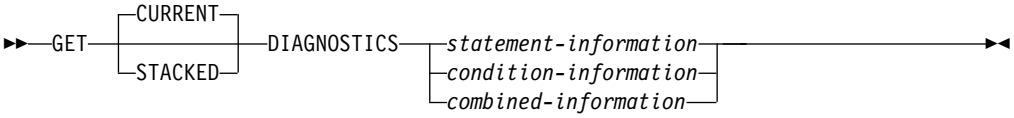
Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It cannot be issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

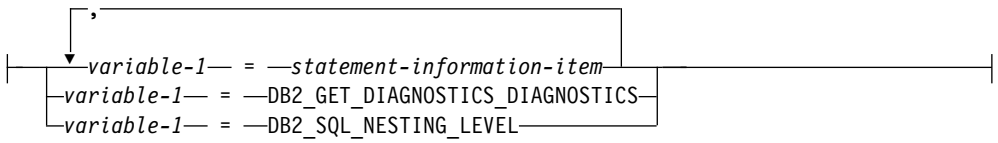
Authorization

None required.

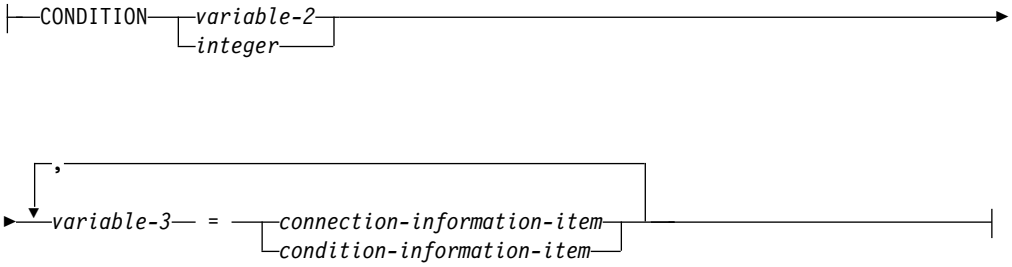
Syntax



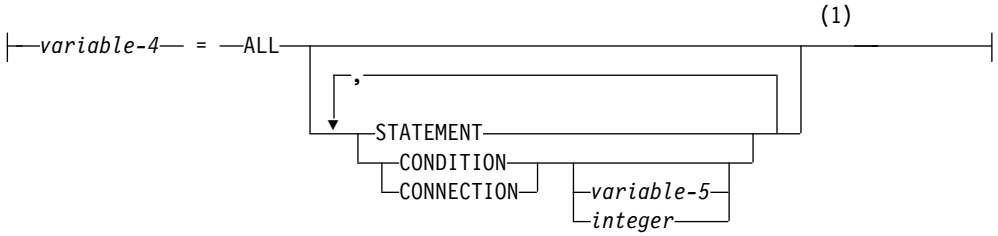
statement-information:



condition-information:



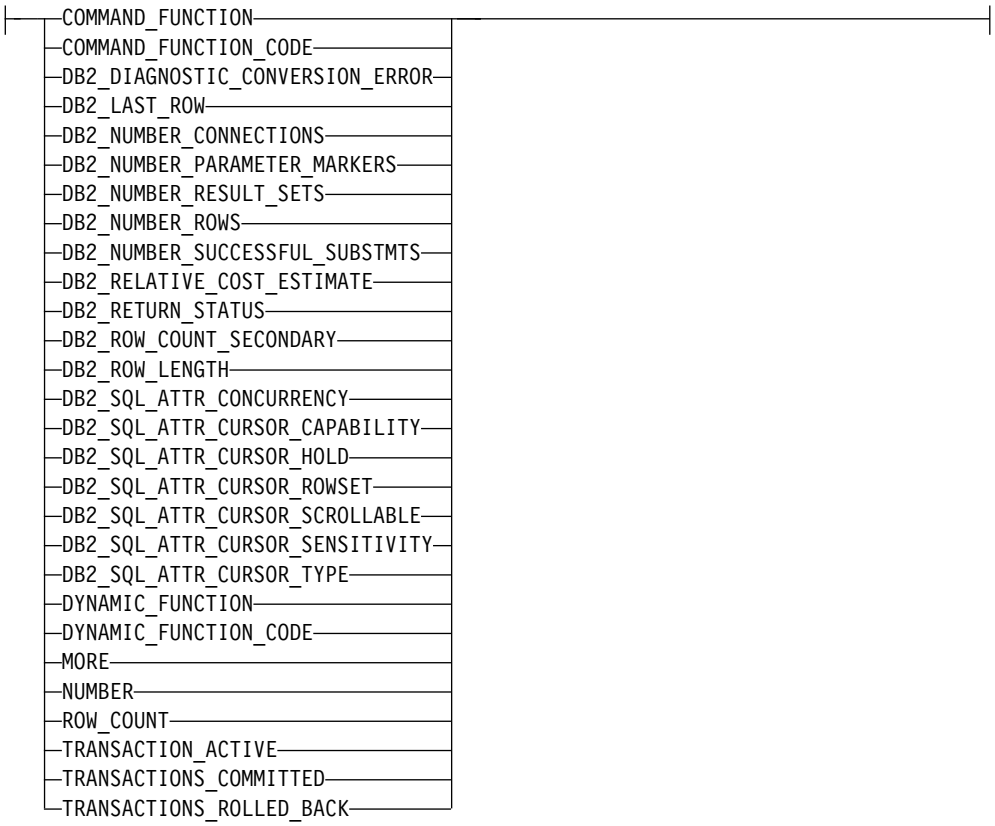
combined-information:



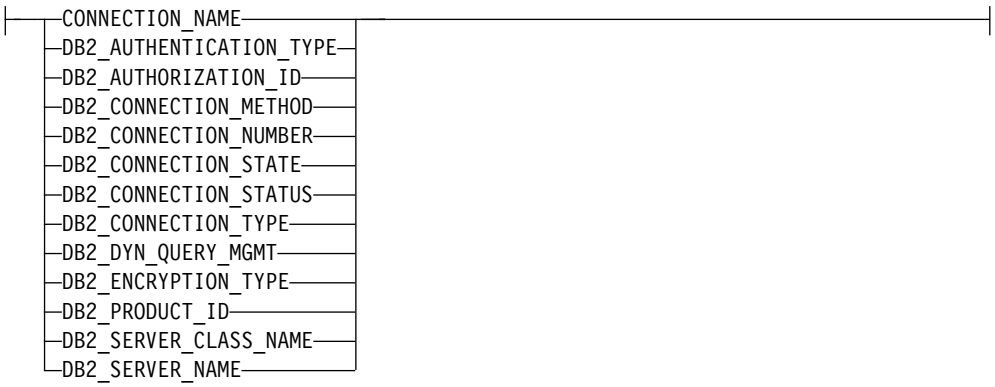
Notes:

1 STATEMENT can only be specified once. If *variable-5* or *integer* is not specified, CONDITION and CONNECTION can only be specified once.

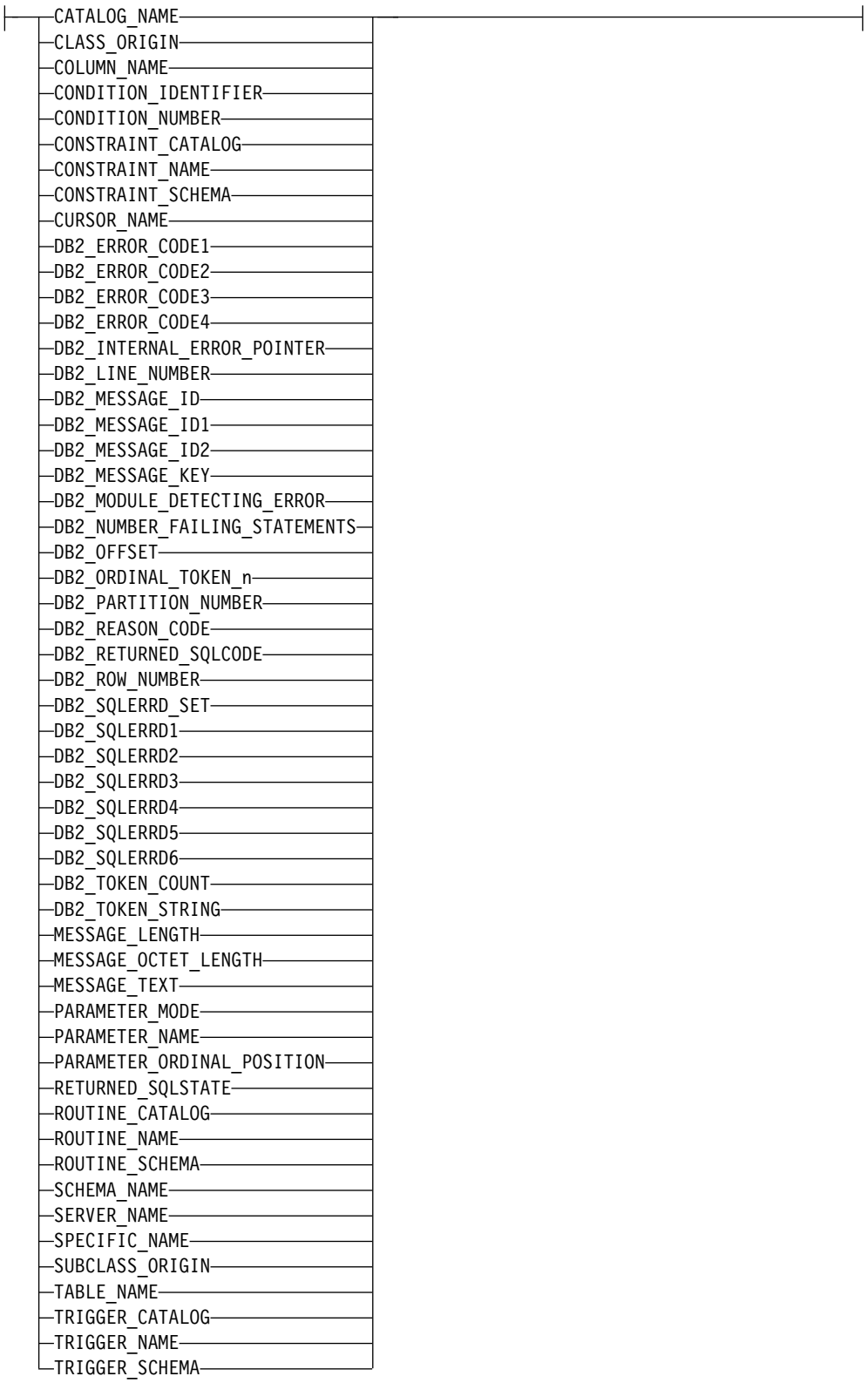
statement-information-item:



connection-information-item:



condition-information-item:



Description

CURRENT or STACKED
Specifies which diagnostics area to access.

CURRENT
Specifies to access the first diagnostics area. It corresponds to the previous SQL statement that was executed and that was not a GET DIAGNOSTICS statement. This is the default.

STACKED
Specifies to access the second diagnostics area. The second diagnostics area is only available within a handler. It corresponds to the previous SQL statement that was executed before the handler was entered and that was not a GET DIAGNOSTICS statement. If the GET DIAGNOSTICS statement is the first statement within a handler, then the first diagnostics area and the second diagnostics area contain the same diagnostics information.

statement-information
Returns information about the last SQL statement executed.

variable–1
Identifies a variable declared in the program in accordance with the rules for declaring variables. It must not be a global variable. The data type of the variable must be compatible with the data type as specified in Table 85 on page 1211 for the specified condition information item. The variable is assigned the value of the specified statement information item according to the retrieval assignment rules described in "Retrieval assignment" on page 102. If the value is truncated when assigning it to the variable, a warning (SQLSTATE 01004) is returned and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition.

If a specified diagnostic item does not contain diagnostic information, then the variable is set to a default value based on its data type:

- 0 for an exact numeric diagnostic item,
- an empty string for a VARCHAR diagnostic item,
- and blanks for a CHAR diagnostic item.

condition-information
Returns information about the condition or conditions that occurred when the last SQL statement was executed.

CONDITION *variable–2* **or** *integer*
Identifies the diagnostic for which information is requested. Each diagnostic that occurs while executing an SQL statement is assigned an integer. The value 1 indicates the first diagnostic, 2 indicates the second diagnostic and so on. If the value is 1, then the diagnostic information retrieved corresponds to the condition indicated by the SQLSTATE value actually returned by the execution of the previous SQL statement (other than a GET DIAGNOSTICS statement). The variable specified must be declared in the program in accordance with the rules for declaring exact numeric variables with zero scale.It must not be a global variable. The value specified must not be less than one or greater than the number of available diagnostics.

variable–3
Identifies a variable declared in the program in accordance with the rules for declaring variables. It must not be a global variable. The data type of the variable must be compatible with the data type as specified in Table 85 on page 1211 for the specified condition information item. The variable is assigned the value of the specified condition information item according to the retrieval assignment rules described in "Retrieval assignment" on page 102. If the value is truncated when assigning it to the variable, an error is

returned and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition.

If a specified diagnostic item does not contain diagnostic information , then the variable is set to a default value based on its data type:

- 0 for an exact numeric diagnostic item,
- an empty string for a VARCHAR diagnostic item,
- and blanks for a CHAR diagnostic item.

combined-information
Returns multiple information items combined into one string.

variable–4
Identifies a variable declared in the program in accordance with the rules for declaring variables. It must not be a global variable. The data type of the variable must be VARCHAR. The variable is assigned according to the retrieval assignment rules described in "Retrieval assignment" on page 102. If the length of *variable–4* is not sufficient to hold the full returned diagnostic string, the string is truncated, an error is returned and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition.

ALL
Indicates that all diagnostic items that are set for the last SQL statement executed should be combined into one string. The format of the string is a semicolon separated list of all of the available diagnostic information in the form:

```
item-name=character-form-of-the-item-value;
```

The character form of a positive numeric value will not contain a leading plus sign (+) unless the item is DB2_RETURNED_SQLCODE. In this case, a leading plus sign (+) is added. For example:

```
NUMBER=1;RETURNED_SQLSTATE=02000;DB2_RETURNED_SQLCODE=+100;
```

Only items that contain diagnostic information are included in the string. There are also no entries in this string for the DB2_GET_DIAGNOSTICS_DIAGNOSTICS and DB2_SQL_NESTING_LEVEL items.

STATEMENT
Indicates that all *statement-information-item* diagnostic items that contain diagnostic information for the last SQL statement executed should be combined into one string. The format is the same as described above for ALL.

CONDITION
Indicates that *condition-information-item* diagnostic items that contain diagnostic information for the last SQL statement executed should be combined into one string. If *variable–5* or *integer* is specified, then the format is the same as described above for the ALL option. If *variable–5* or *integer* is not specified, then the format includes a condition number entry at the beginning of the information for that condition in the form:

```
CONDITION_NUMBER=X;item-name=character-form-of-the-item-value;
```

where X is the number of the condition. For example:

```
CONDITION_NUMBER=1;RETURNED_SQLSTATE=02000;DB2_RETURNED_SQLCODE=+100;
CONDITION_NUMBER=2;RETURNED_SQLSTATE=01004;
```

CONNECTION
Indicates that *connection-information-item* diagnostic items that contain diagnostic information for the last SQL statement executed should be combined into one string. If *variable–5* or *integer* is specified, then the format is the same as described above for ALL. If *variable–5* or *integer* is not specified, then the format includes a connection number entry at the beginning of the information for that condition in the form:

```
DB2_CONNECTION_NUMBER=X;item-name=character-form-of-the-item-value;
```

where X is the number of the condition. For example:

```
DB2_CONNECTION_NUMBER=1;CONNECTION_NAME=SVL1;DB2_PRODUCT_ID=DSN07010;
```

variable–5 **or** *integer*
Identifies the diagnostic for which ALL CONDITION or ALL CONNECTION information is requested. The variable specified must be declared in the program in accordance with the rules for declaring integer variables. It must not be a global variable. The value specified must not be less than one or greater than the number of available diagnostics.

statement-information-item

COMMAND_FUNCTION
Returns the name of the previous SQL statement. For information about the statement string values, see Table 86 on page 1214.

COMMAND_FUNCTION_CODE
Returns an integer that identifies the previous SQL statement. For information about the statement code values, see Table 86 on page 1214.

DB2_DIAGNOSTIC_CONVERSION_ERROR
Returns the value 1 if there was a conversion error when converting a character data value for one of the GET DIAGNOSTICS statement values. Otherwise, the value zero is returned.

DB2_GET_DIAGNOSTICS_DIAGNOSTICS
After a GET DIAGNOSTICS statement, if any errors or warnings occurred during the execution of the GET DIAGNOSTICS statement, DB2_GET_DIAGNOSTICS_DIAGNOSTICS returns textual information about these errors or warnings. The format of the information is similar to what would be returned by a GET DIAGNOSTICS :hv = ALL statement.

If a request was made for an information item that the server does not understand, for example, if the server was at a lower DRDA level than the requesting client, DB2_GET_DIAGNOSTICS_DIAGNOSTICS returns the text 'Item not supported:' followed by a comma separated list of item names that were requested but that the server does not support.

DB2_LAST_ROW
For a *multiple-row-fetch* statement, a value of +100 might be returned if the set of rows that have been fetched contains the last row currently in the result table for cursors that are fetching forward, or contains the first row currently in the result table, for cursors that are fetching backward. If the number of rows returned is equal to the number of rows requested, then an end of data warning may not occur and DB2_LAST_ROW may not contain +100.

If a value of +100 for DB2_LAST_ROW is returned for a cursor that is not sensitive to updates, a subsequent FETCH would return with the SQLCODE set to +100 and SQLSTATE set to '02000'. For a cursor that is sensitive to

updates, a subsequent FETCH might return more data if any rows have been inserted before the FETCH was executed.

For statements other than *multiple-row-fetch* statements, for *multiple-row-fetch* statements that do not contain the last row currently in the result table, for cursors that are fetching forwards, or that do not contain the first row currently in the result table, for cursors that are fetching backwards, or if the server only returns an SQLCA, the value zero is returned.

DB2_NUMBER_CONNECTIONS
Returns the number of connections that were made in order to get to the server that fulfilled the request from the client. Each such connection may generate a connection information item area which would be available for the single condition.

DB2_NUMBER_PARAMETER_MARKERS
For a PREPARE statement, returns the number of parameter markers in the prepared statement. Otherwise, the value zero is returned.

DB2_NUMBER_RESULT_SETS
For a CALL statement, returns the actual number of result sets returned by the procedure. Otherwise, the value zero is returned.

DB2_NUMBER_ROWS
If the previous SQL statement was an OPEN or a FETCH which caused the size of the result table to be known, returns the number of rows in the result table. For SENSITIVE cursors, this value can be thought of as an approximation since rows inserted and deleted will affect the next retrieval of this value. Otherwise, the value zero is returned.

DB2_NUMBER_SUCCESSFUL_SUBSTMTS
For embedded compound SQL statements, returns a count of the number of successful sub-statements. Otherwise, the value zero is returned.

DB2_RELATIVE_COST_ESTIMATE
For a PREPARE statement, returns a relative cost estimate of the resources required for every execution. It does not reflect an estimate of the time required. When preparing a dynamically defined statement, this value can be used as an indicator of the relative cost of the prepared statement. The value varies depending on changes to statistics and can vary between releases of the product. It is an estimated cost for the access plan chosen by the optimizer. The value zero is returned if the statement is not a PREPARE statement.

DB2_RETURN_STATUS
Identifies the status value returned from the previous SQL CALL statement. If the previous statement is not a CALL statement, the value returned has no meaning and is unpredictable. For more information, see "RETURN statement" on page 1483. Otherwise, the value zero is returned.

For external procedures, if the returned SQLCODE < 0, the SQL_ERROR_CODE1 and DB2_RETURN_STATUS will be set to -1, otherwise SQL_ERROR_CODE1 and DB2_RETURN_STATUS are set to 0.

DB2_ROW_COUNT_SECONDARY
Identifies the number of rows associated with secondary actions from the previous SQL statement that was executed. If the previous SQL statement is a DELETE or MERGE, the value is the total number of rows affected by referential constraints, including cascaded actions and the processing of triggered SQL statements from activated triggers. If the previous SQL statement is an INSERT or an UPDATE, the value is the total number of rows

affected as the result of the processing of triggered SQL statements from activated triggers. Otherwise, the value zero is returned.

If the SQL statement is run using isolation level No Commit, this value may be zero.

DB2_ROW_LENGTH
For FETCH, if the result row does not contain a LOB, returns the length of the row(s) retrieved. For OPEN, if the result row does not contain a LOB, returns the length of a result row. For FETCH and OPEN, if the result does contain a LOB, the length returned is unpredictable. Otherwise, the value zero is returned.

DB2_SQL_ATTR_CONCURRENCY
For an OPEN statement, indicates the concurrency control option of read-only, locking, optimistic using timestamps, or optimistic using values.

- R indicates read-only.
- L indicates locking.
- T indicates comparing row versions using timestamps or ROWIDs.
- V indicates comparing values.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_CAPABILITY
For an OPEN statement, indicates the capability of the cursor, whether a cursor is read-only, deletable, or updatable.

- R indicates that this cursor can only be used to read.
- D indicates that this cursor can be used to read as well as delete.
- U indicates that this cursor can be used to read, delete as well as update.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_HOLD
For an OPEN statement, indicates whether a cursor can be held open across multiple units of work or not.

- N indicates that this cursor will not remain open across multiple units of work.
- Y indicates that this cursor will remain open across multiple units of work.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_ROWSET
For an OPEN statement, whether a cursor can be accessed using rowset positioning or not.

- N indicates that this cursor only supports row positioned operations.
- Y indicates that this cursor supports rowset positioned operations.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_SCROLLABLE
For an OPEN statement, indicates whether a cursor can be scrolled forward and backward or not.

- N indicates that this cursor is not scrollable.
- Y indicates that this cursor is scrollable.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_SENSITIVITY
For an OPEN statement, indicates whether a cursor does or does not show updates to cursor rows made by other connections.

- I indicates insensitive.
- P indicates partial sensitivity.
- S indicates sensitive.
- U indicates unspecified.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_TYPE
For an OPEN statement, indicates whether a cursor type is dynamic, forward-only, or static.

- D indicates a dynamic cursor.
- F indicates a forward-only cursor.
- S indicates a static cursor.

Otherwise, a blank is returned.

DB2_SQL_NESTING_LEVEL
Identifies the current level of nesting or recursion in effect when the GET DIAGNOSTICS statement was executed. Each level of nesting corresponds to a nested or recursive invocation of a function, procedure, or trigger. If the GET DIAGNOSTICS statement is executed outside of a level of nesting, the value zero is returned.

If GET DIAGNOSTIC is issued in a user-defined function that is running in parallel, the nesting level is not predictable.

DYNAMIC_FUNCTION
Returns a character string that identifies the type of the SQL-statement being prepared or executed dynamically. For information about the statement string values, see Table 86 on page 1214.

DYNAMIC_FUNCTION_CODE
Returns a number that identifies the type of the SQL-statement being prepared or executed dynamically. For information about the statement code values, see Table 86 on page 1214.

MORE
Indicates whether more errors were raised than could be handled.

- N indicates that all the errors and warnings from the previous SQL statement were stored in the diagnostics area.
- Y indicates that more errors and warnings were raised from the previous SQL statement than there are condition areas in the diagnostics area. The maximum size of the diagnostics area is 90K.

NUMBER
Returns the number of errors and warnings detected by the execution of the previous SQL statement, other than a GET DIAGNOSTICS statement, that have been stored in the diagnostics area. If the previous SQL statement returned success (SQLSTATE 00000), or no previous SQL statement has been executed, the number returned is one. The GET DIAGNOSTICS statement itself may return information via the SQLSTATE parameter, but does not modify the previous contents of the diagnostics area, except for the DB2_GET_DIAGNOSTICS_DIAGNOSTICS item.

ROW_COUNT
Identifies the number of rows associated with the previous SQL statement that

was executed. If the previous SQL statement is a DELETE, INSERT, REFRESH, or UPDATE statement, ROW_COUNT identifies the number of rows deleted, inserted, or updated by that statement, excluding rows affected by either triggers or referential integrity constraints. If the previous SQL statement is a MERGE statement, ROW_COUNT identifies the total number of rows deleted, inserted, and updated by that statement, excluding rows affected by either triggers or referential integrity constraints. If the previous SQL statement is a *multiple-row-fetch*, ROW_COUNT identifies the number of rows fetched. Otherwise, the value zero is returned.

TRANSACTION_ACTIVE
Returns the value 1 if an SQL transaction is currently active, and 0 if an SQL transaction is not currently active.

TRANSACTIONS_COMMITTED
If the previous statement was a CALL, returns the number of transactions that were committed during the execution of the SQL or external procedure. Otherwise, the value zero is returned.

TRANSACTIONS_ROLLED_BACK
If the previous statement was a CALL, returns the number of transactions that were rolled back during the execution of the SQL or external procedure. Otherwise, the value zero is returned.

connection-information-item

CONNECTION_NAME
If the previous SQL statement is a CONNECT, DISCONNECT, or SET CONNECTION, returns the name of the server specified in the previous statement. Otherwise, the name of the current connection.

DB2_AUTHENTICATION_TYPE
Indicates the authentication type, whether server or client.

- C for client authentication.
- E for DCE security services authentication.
- S for server authentication.

Otherwise, a blank is returned.

DB2_AUTHORIZATION_ID
Returns the authorization id used by connected server. Because of userid translation and authorization exits, the local userid may not be the authid used by the server.

DB2_CONNECTION_METHOD
For a CONNECT or SET CONNECTION statement, returns the connection method.

- D indicates *DUW (Distributed Unit of Work).
- R indicates *RUW (Remote Unit of Work).

DB2_CONNECTION_NUMBER
Returns the number of the connections.

DB2_CONNECTION_STATE
Indicates the connection state, whether connected or not.

- -1 indicates the connection is unconnected.
- 1 indicates the connection is connected.

Otherwise, the value zero is returned.

DB2_CONNECTION_STATUS
Indicates whether committable update can be performed or not.

- 1 indicates committable updates can be performed on the connection for this unit of work.
- 2 indicates no committable updates can be performed on the connection for this unit of work.

Otherwise, the value zero is returned.

DB2_CONNECTION_TYPE
Indicated the connection type (either local, remote, or to a driver program) and whether the conversation is protected or not.

- 1 indicates a connection to a local relational database.
- 2 indicates a connection to a remote relational database with the conversation unprotected.
- 3 indicates a connection to a remote relational database with the conversation protected.
- 4 indicates a connection to an application requester driver program.

Otherwise, the value zero is returned.

DB2_DYN_QUERY_MGMT
Returns a value of 1 if DYN_QUERY_MGMT database configuration parameter is enabled. Otherwise, the value zero is returned.

DB2_ENCRYPTION_TYPE
Returns the level of encryption.

- A indicates only the authentication tokens (authid and password) are encrypted.
- D indicates all data is encrypted for the connection.

Otherwise, a blank is returned.

DB2_PRODUCT_ID
Returns a product signature. If the application server is an IBM relational database product, the form is pppvvrrm, where:

- ppp identifies the product as follows: ARI for DB2 for VM and VSE, DSN for DB2 for z/OS, QSQ for DB2 for i, and SQL for all other DB2 products
- vv is a two-digit version identifier such as '04'
- rr is a two-digit release identifier such as '01'
- m is a one-digit modification level such as '0'

For example, if the application server is Version 7 of DB2 for z/OS, the value would be 'DSN07010'. Otherwise, the empty string is returned.

DB2_SERVER_CLASS_NAME
Returns the server class name. For example, DB2 for z/OS, DB2 for AIX, DB2 for Windows, and DB2 for i.

DB2_SERVER_NAME
For a CONNECT or SET CONNECTION statement, returns the relational database name. Otherwise, the empty string is returned.

condition-information-item

CATALOG_NAME
If the returned SQLSTATE is:

- class 09 (Triggered Action Exception), or

- class 23 (Integrity Constraint Violation), or
- class 27 (Triggered Data Change Violation), or
- 40002 (Transaction Rollback - Integrity Constraint Violation),

and the constraint that caused the error is a referential, check, or unique constraint, the server name of the table that owns the constraint is returned.

If the returned SQLSTATE is class 42 (Syntax Error or Access Rule Violation), the server name of the table that caused the error is returned.

If the returned SQLSTATE is class 44 (WITH CHECK OPTION Violation), the server name of the view that caused the error is returned. Otherwise, the empty string is returned.

CLASS_ORIGIN
Returns 'ISO 9075' for those SQLSTATEs whose class is defined by ISO 9075. Returns 'ISO/IEC 13249' for those SQLSTATEs whose class is defined by SQL/MM. Returns 'DB2 SQL' for those SQLSTATEs whose class is defined by IBM DB2 SQL. Returns the value set by user written code if available. Otherwise, the empty string is returned.

COLUMN_NAME
If the returned SQLSTATE is class 42 (Syntax Error or Access Rule Violation) and the error was caused by an inaccessible column, the name of the column that caused the error is returned. Otherwise, the empty string is returned.

CONDITION_IDENTIFIER
If the value of the RETURNED_SQLSTATE corresponds to an unhandled user-defined exception (SQLSTATE 45000), then the condition name of the user-defined exception is returned.

CONDITION_NUMBER
Returns the number of the conditions.

CONSTRAINT_CATALOG
If the returned SQLSTATE is:

- class 23 (Integrity Constraint Violation), or
- class 27 (Triggered Data Change Violation), or
- 40002 (Transaction Rollback - Integrity Constraint Violation),

the name of the server that contains the table that contains the constraint that caused the error is returned. Otherwise, the empty string is returned.

CONSTRAINT_NAME
If the returned SQLSTATE is:

- class 23 (Integrity Constraint Violation), or
- class 27 (Triggered Data Change Violation), or
- 40002 (Transaction Rollback - Integrity Constraint Violation),

the name of the constraint that caused the error is returned. Otherwise, the empty string is returned.

CONSTRAINT_SCHEMA
If the returned SQLSTATE is:

- class 23 (Integrity Constraint Violation), or
- class 27 (Triggered Data Change Violation), or
- 40002 (Transaction Rollback - Integrity Constraint Violation),

the name of the schema of the constraint that caused the error is returned. Otherwise, the empty string is returned.

CURSOR_NAME
If the returned SQLSTATE is class 24 (Invalid Cursor State), the name of the cursor is returned. Otherwise, the empty string is returned.

DB2_ERROR_CODE1
Returns an internal error code. Otherwise, the value zero is returned.

DB2_ERROR_CODE2
Returns an internal error code. Otherwise, the value zero is returned.

DB2_ERROR_CODE3
Returns an internal error code. Otherwise, the value zero is returned.

DB2_ERROR_CODE4
Returns an internal error code. Otherwise, the value zero is returned.

DB2_INTERNAL_ERROR_POINTER
For some errors, this will be a negative value that is an internal error pointer. Otherwise, the value zero is returned.

DB2_LINE_NUMBER
For a CREATE PROCEDURE for an SQL function, SQL procedure, or SQL trigger where an error is encountered parsing the SQL procedure body, returns the line number where the error possibly occurred. Otherwise, the value zero is returned.

DB2_MESSAGE_ID
Returns the message ID corresponding to the MESSAGE_TEXT.

DB2_MESSAGE_ID1
Returns the underlying IBM i CPF escape message that originally caused this error. Otherwise, the empty string is returned.

DB2_MESSAGE_ID2
Returns the underlying IBM i CPD diagnostic message that originally caused this error. Otherwise, the empty string is returned.

DB2_MESSAGE_KEY
For a CALL statement, returns the IBM i message key of the error that caused the procedure to fail. For a trigger error in a DELETE, INSERT, or UPDATE statement, returns the message key of the error that was signaled from the trigger program. The IBM i QMHRCVPM API can be used to return the message description and message data for the message key. Otherwise, the value zero is returned.

DB2_MODULE_DETECTING_ERROR
Returns an identifier indicating which module detected the error. For a SIGNAL statement issued from a routine, the value 'ROUTINE' is returned. For other SIGNAL statements, the value 'PROGRAM' is returned.

DB2_NUMBER_FAILING_STATEMENTS
For a NOT ATOMIC embedded compound SQL statement, returns the number of statements that failed. Otherwise, the value zero is returned.

DB2_OFFSET
For a CREATE PROCEDURE for an SQL procedure where an error is encountered parsing the SQL procedure body, returns the offset into the line number where the error possibly occurred, if available. For an EXECUTE IMMEDIATE or a PREPARE statement where an error is encountered parsing the source statement, returns the offset into the source statement where the error possibly occurred. Otherwise, the value zero is returned.

DB2_ORDINAL_TOKEN_n
Returns the nth token. n must be a value from 1 to 100. For example, DB2_ORDINAL_TOKEN_1 would return the value of the first token, DB2_ORDINAL_TOKEN_2 the second token. A numeric value for a token is converted to character before being returned. If there is no value for the token, the empty string is returned.

DB2_PARTITION_NUMBER
For a partitioned database, returns the partition number of the database partition that encountered the error or warning. If no errors or warnings were encountered, returns the partition number of the current node. Otherwise, the value zero is returned.

DB2_REASON_CODE
Returns the reason code for errors that have a reason code token in the message text. Otherwise, the value zero is returned.

DB2_RETURNED_SQLCODE
Returns the SQLCODE for the specified diagnostic.

DB2_ROW_NUMBER
If the previous SQL statement is a multiple row insert or a multiple row fetch, returns the number of the row where the condition was encountered, when such a value is available and applicable. Otherwise, the value zero is returned.

DB2_SQLERRD_SET
Returns Y to indicate that the DB2_SQLERRD1 through DB2_SQLERRD6 items may be set. Otherwise, a blank is returned.

DB2_SQLERRD1
Returns the value of SQLERRD(1) from the SQLCA returned by the server.

DB2_SQLERRD2
Returns the value of SQLERRD(2) from the SQLCA returned by the server.

DB2_SQLERRD3
Returns the value of SQLERRD(3) from the SQLCA returned by the server.

DB2_SQLERRD4
Returns the value of SQLERRD(4) from the SQLCA returned by the server.

DB2_SQLERRD5
Returns the value of SQLERRD(5) from the SQLCA returned by the server.

DB2_SQLERRD6
Returns the value of SQLERRD(6) from the SQLCA returned by the server.

DB2_TOKEN_COUNT
Returns the number of tokens available for the specified diagnostic.

DB2_TOKEN_STRING
Returns a X'FF' delimited string of the tokens for the specified diagnostic.

MESSAGE_LENGTH
Identifies the length (in characters) of the message text of the error, warning, or successful completion returned from the previous SQL statement that was executed.

MESSAGE_OCTET_LENGTH
Identifies the length (in bytes) of the message text of the error, warning, or successful completion returned from the previous SQL statement that was executed.

MESSAGE_TEXT
Identifies the message text of the error, warning, or successful completion returned from the previous SQL statement that was executed.

When the SQLCODE is 0, the empty string is returned, even if the RETURNED_SQLSTATE value indicates a warning condition.

PARAMETER_MODE
If the returned SQLSTATE is:

- class 39 (External Routine Invocation Exception), or
- class 38 (External Routine Exception), or
- class 2F (SQL Routine Exception), or
- class 22 (Data Exception), or
- class 23 (Integrity Constraint Violation), or
- class 01 (Warning)

and the condition is related to the *i*th parameter of the routine, the parameter mode of the *i*th parameter is returned. Otherwise, the empty string is returned.

PARAMETER_NAME
If the returned SQLSTATE is:

- class 39 (External Routine Invocation Exception), or
- class 38 (External Routine Exception), or
- class 2F (SQL Routine Exception), or
- class 22 (Data Exception), or
- class 23 (Integrity Constraint Violation), or
- class 01 (Warning)

the condition is related to the *i*th parameter of the routine, and a parameter name was specified for the parameter when the routine was created, the parameter name of the *i*th parameter is returned. Otherwise, the empty string is returned.

PARAMETER_ORDINAL_POSITION
If the returned SQLSTATE is:

- class 39 (External Routine Invocation Exception), or
- class 38 (External Routine Exception), or
- class 2F (SQL Routine Exception), or
- class 22 (Data Exception), or
- class 23 (Integrity Constraint Violation), or
- class 01 (Warning)

and the condition is related to the *i*th parameter of the routine, the value of *i* is returned. Otherwise, the empty string is returned.

RETURNED_SQLSTATE
Returns the SQLSTATE for the specified diagnostic.

ROUTINE_CATALOG
If the returned SQLSTATE is:

- class 39 (External Routine Invocation Exception), or
- class 38 (External Routine Exception), or
- class 2F (SQL Routine Exception), or

and the condition is related to the *i*th parameter of the routine, or if the returned SQLSTATE is:

- class 22 (Data Exception), or
- class 23 (Integrity Constraint Violation), or
- class 01 (Warning)

and the condition was raised as the result of an assignment to an SQL parameter during an routine invocation, the server name of the routine is returned. Otherwise, the empty string is returned.

ROUTINE_NAME

If the returned SQLSTATE is:

- class 39 (External Routine Invocation Exception), or
- class 38 (External Routine Exception), or
- class 2F (SQL Routine Exception), or

and the condition is related to the *i*th parameter of the routine, or if the returned SQLSTATE is:

- class 22 (Data Exception), or
- class 23 (Integrity Constraint Violation), or
- class 01 (Warning)

and the condition was raised as the result of an assignment to an SQL parameter during an routine invocation, the name of the routine is returned. Otherwise, the empty string is returned.

ROUTINE_SCHEMA

If the returned SQLSTATE is:

- class 39 (External Routine Invocation Exception), or
- class 38 (External Routine Exception), or
- class 2F (SQL Routine Exception), or

and the condition is related to the *i*th parameter of the routine, or if the returned SQLSTATE is:

- class 22 (Data Exception), or
- class 23 (Integrity Constraint Violation), or
- class 01 (Warning)

and the condition was raised as the result of an assignment to an SQL parameter during an routine invocation, the schema name of the routine is returned. Otherwise, the empty string is returned.

SCHEMA_NAME

If the returned SQLSTATE is:

- class 09 (Triggered Action Exception), or
- class 23 (Integrity Constraint Violation), or
- class 27 (Triggered Data Change Violation), or
- 40002 (Transaction Rollback - Integrity Constraint Violation),

and the constraint that caused the error is a referential, check, or unique constraint, the schema name of the table that owns the constraint is returned.

If the returned SQLSTATE is class 42 (Syntax Error or Access Rule Violation), the schema name of the table that caused the error is returned.

If the returned SQLSTATE is class 44 (WITH CHECK OPTION Violation), the schema name of the view that caused the error is returned. Otherwise, the empty string is returned.

SERVER_NAME
If the previous SQL statement is a CONNECT, DISCONNECT, or SET CONNECTION, the name of the server specified in the previous statement is returned. Otherwise, the name of the server where the statement executed is returned.

SPECIFIC_NAME
If the returned SQLSTATE is:

- class 39 (External Routine Invocation Exception), or
- class 38 (External Routine Exception), or
- class 2F (SQL Routine Exception), or

and the condition is related to the *i*th parameter of the routine, or if the returned SQLSTATE is:

- class 22 (Data Exception), or
- class 23 (Integrity Constraint Violation), or
- class 01 (Warning)

and the condition was raised as the result of an assignment to an SQL parameter during an routine invocation, the specific name of the procedure or function is returned. Otherwise, the empty string is returned.

SUBCLASS_ORIGIN
Returns 'ISO 9075' for those SQLSTATEs whose subclass is defined by ISO 9075. Returns 'ISO/IEC 9579' for those SQLSTATEs whose subclass is defined by RDA. Returns 'ISO/IEC 13249-1', 'ISO/IEC 13249-2', 'ISO/IEC 13249-3', 'ISO/IEC 13249-4', or 'ISO/IEC 13249-5' for those SQLSTATEs whose subclass is defined SQL/MM. Returns 'DB2 SQL' for those SQLSTATEs whose subclass is defined by IBM DB2 SQL. Returns the value set by user written code if available. Otherwise, the empty string is returned.

TABLE_NAME
If the returned SQLSTATE is:

- class 09 (Triggered Action Exception), or
- class 23 (Integrity Constraint Violation), or
- class 27 (Triggered Data Change Violation), or
- 40002 (Transaction Rollback - Integrity Constraint Violation),

and the constraint that caused the error is a referential, check, or unique constraint, the table name that owns the constraint is returned.

If the returned SQLSTATE is class 42 (Syntax Error or Access Rule Violation), the table name that caused the error is returned.

If the returned SQLSTATE is class 44 (WITH CHECK OPTION Violation), the table name that caused the error is returned. Otherwise, the empty string is returned.

TRIGGER_CATALOG
If the returned SQLSTATE is:

- class 09 (Triggered Action Exception), or
- class 27 (Triggered Data Change Violation),

the name of the trigger is returned. Otherwise, the empty string is returned.

TRIGGER_NAME
If the returned SQLSTATE is:

- class 09 (Triggered Action Exception), or

- class 27 (Triggered Data Change Violation),

the name of the trigger is returned. Otherwise, the empty string is returned.

TRIGGER_SCHEMA
If the returned SQLSTATE is:

- class 09 (Triggered Action Exception), or
- class 27 (Triggered Data Change Violation),

the schema name of the trigger is returned. Otherwise, the empty string is returned.

Notes

Considerations for the diagnostics area: The GET DIAGNOSTICS statement does not change the contents of the diagnostics area except for DB2_GET_DIAGNOSTICS_DIAGNOSTICS.

If the GET DIAGNOSTICS statement is specified in an SQL function, SQL procedure, or trigger:

- If information is desired about an error, the GET DIAGNOSTICS statement must be the first executable statement specified in the handler that will handle the error.
- If information is wanted about a warning:
  - If a handler will get control for the warning condition, the GET DIAGNOSTICS statement must be the first statement specified in that handler.
  - If a handler will not get control for the warning condition, the GET DIAGNOSTICS statement must be the next statement executed after that previous statement.

Otherwise, GET DIAGNOSTICS statement returns information about the last executed statement.

Considerations for the SQLCODE and SQLSTATE SQL variables: The GET DIAGNOSTICS statement does not change the value of the SQLSTATE and SQLCODE SQL variables.

Case of return values: Values for identifiers in returned diagnostic items are not delimited and are case sensitive. For example, a table name of "abc" would be returned, simply as abc.

Variable assignment: If an assignment error occurs, the values in the variables are unpredictable.

Data types for items: The following table shows, the SQL data type for each diagnostic item. When a diagnostic item is assigned to a variable, the variable must be compatible with the data type of the diagnostic item.

Table 85. Data Types for GET DIAGNOSTICS Items

Item Name	Data Type
Statement Information Item	
COMMAND_FUNCTION	VARCHAR(128)
COMMAND_FUNCTION_CODE	INTEGER

Table 85. Data Types for GET DIAGNOSTICS Items (continued)

Item Name	Data Type
DB2_DIAGNOSTIC_CONVERSION_ERROR	INTEGER
DB2_GET_DIAGNOSTICS_DIAGNOSTICS	VARCHAR(32740)
DB2_LAST_ROW	INTEGER
DB2_NUMBER_CONNECTIONS	INTEGER
DB2_NUMBER_PARAMETER_MARKERS	INTEGER
DB2_NUMBER_RESULT_SETS	INTEGER
DB2_NUMBER_ROWS	DECIMAL(31,0)
DB2_NUMBER_SUCCESSFUL_SUBSTMTS	INTEGER
DB2_RELATIVE_COST_ESTIMATE	INTEGER
DB2_RETURN_STATUS	INTEGER
DB2_ROW_COUNT_SECONDARY	DECIMAL(31,0)
DB2_ROW_LENGTH	INTEGER
DB2_SQL_ATTR_CONCURRENCY	CHAR(1)
DB2_SQL_ATTR_CURSOR_CAPABILITY	CHAR(1)
DB2_SQL_ATTR_CURSOR_HOLD	CHAR(1)
DB2_SQL_ATTR_CURSOR_ROWSET	CHAR(1)
DB2_SQL_ATTR_CURSOR_SCROLLABLE	CHAR(1)
DB2_SQL_ATTR_CURSOR_SENSITIVITY	CHAR(1)
DB2_SQL_ATTR_CURSOR_TYPE	CHAR(1)
DB2_SQL_NESTING_LEVEL	INTEGER
DYNAMIC_FUNCTION	VARCHAR(128)
DYNAMIC_FUNCTION_CODE	INTEGER
MORE	CHAR(1)
NUMBER	INTEGER
ROW_COUNT	DECIMAL(31,0)
TRANSACTION_ACTIVE	INTEGER
TRANSACTIONS_COMMITTED	INTEGER
TRANSACTIONS_ROLLED_BACK	INTEGER
Connection Information Item	
CONNECTION_NAME	VARCHAR(128)
DB2_AUTHENTICATION_TYPE	CHAR(1)
DB2_AUTHORIZATION_ID	VARCHAR(128)
DB2_CONNECTION_METHOD	CHAR(1)
DB2_CONNECTION_NUMBER	INTEGER
DB2_CONNECTION_STATE	INTEGER
DB2_CONNECTION_STATUS	INTEGER
DB2_CONNECTION_TYPE	SMALLINT
DB2_DYN_QUERY_MGMT	INTEGER
DB2_ENCRYPTION_TYPE	CHAR(1)
DB2_PRODUCT_ID	VARCHAR(8)

Table 85. Data Types for GET DIAGNOSTICS Items (continued)

Item Name	Data Type
DB2_SERVER_CLASS_NAME	VARCHAR(128)
DB2_SERVER_NAME	VARCHAR(128)
Condition Information Item	
CATALOG_NAME	VARCHAR(128)
CLASS_ORIGIN	VARCHAR(128)
COLUMN_NAME	VARCHAR(128)
CONDITION_IDENTIFIER	VARCHAR(128)
CONDITION_NUMBER	INTEGER
CONSTRAINT_CATALOG	VARCHAR(128)
CONSTRAINT_NAME	VARCHAR(128)
CONSTRAINT_SCHEMA	VARCHAR(128)
CURSOR_NAME	VARCHAR(128)
DB2_ERROR_CODE1	INTEGER
DB2_ERROR_CODE2	INTEGER
DB2_ERROR_CODE3	INTEGER
DB2_ERROR_CODE4	INTEGER
DB2_INTERNAL_ERROR_POINTER	INTEGER
DB2_LINE_NUMBER	INTEGER
DB2_MESSAGE_ID	CHAR(10)
DB2_MESSAGE_ID1	VARCHAR(7)
DB2_MESSAGE_ID2	VARCHAR(7)
DB2_MESSAGE_KEY	INTEGER
DB2_MODULE_DETECTING_ERROR	VARCHAR(128)
DB2_NUMBER_FAILING_STATEMENTS	INTEGER
DB2_OFFSET	INTEGER
DB2_ORDINAL_TOKEN_n	VARCHAR(32740)
DB2_PARTITION_NUMBER	INTEGER
DB2_REASON_CODE	INTEGER
DB2_RETURNED_SQLCODE	INTEGER
DB2_ROW_NUMBER	INTEGER
DB2_SQLERRD_SET	CHAR(1)
DB2_SQLERRD1	INTEGER
DB2_SQLERRD2	INTEGER
DB2_SQLERRD3	INTEGER
DB2_SQLERRD4	INTEGER
DB2_SQLERRD5	INTEGER
DB2_SQLERRD6	INTEGER
DB2_TOKEN_COUNT	INTEGER
DB2_TOKEN_STRING	VARCHAR(1000)
MESSAGE_LENGTH	INTEGER

Table 85. Data Types for GET DIAGNOSTICS Items (continued)

Item Name	Data Type
MESSAGE_OCTET_LENGTH	INTEGER
MESSAGE_TEXT	VARCHAR(32740)
PARAMETER_MODE	VARCHAR(5)
PARAMETER_NAME	VARCHAR(128)
PARAMETER_ORDINAL_POSITION	INTEGER
RETURNED_SQLSTATE	CHAR(5)
ROUTINE_CATALOG	VARCHAR(128)
ROUTINE_NAME	VARCHAR(128)
ROUTINE_SCHEMA	VARCHAR(128)
SCHEMA_NAME	VARCHAR(128)
SERVER_NAME	VARCHAR(128)
SPECIFIC_NAME	VARCHAR(128)
SUBCLASS_ORIGIN	VARCHAR(128)
TABLE_NAME	VARCHAR(128)
TRIGGER_CATALOG	VARCHAR(128)
TRIGGER_NAME	VARCHAR(128)
TRIGGER_SCHEMA	VARCHAR(128)

SQL statement codes and strings: The following table represents the possible values for COMMAND_FUNCTION, COMMAND_FUNCTION_CODE, DYNAMIC_FUNCTION, and DYNAMIC_FUNCTION_CODE diagnostic items.

The values in the following table are assigned by the ISO and ANSI SQL Standard and may change as the standard evolves. Include *sqlscds* in the include source files in library QSYSINC should be used when referencing these values.

Table 86. SQL Statement Codes and Strings

Type of statement	Statement string	Statement code
ALLOCATE CURSOR	ALLOCATE CURSOR	1
ALLOCATE DESCRIPTOR	ALLOCATE DESCRIPTOR	2
ALTER FUNCTION	ALTER ROUTINE	17
ALTER PROCEDURE	ALTER ROUTINE	17
ALTER SEQUENCE	ALTER SEQUENCE	134
ALTER TABLE	ALTER TABLE	4
assignment-statement	ASSIGNMENT	5
ASSOCIATE LOCATORS	ASSOCIATE LOCATORS	–6
CALL	CALL	7
CASE	CASE	86
CLOSE (static SQL)	CLOSE CURSOR	9
CLOSE (dynamic SQL)	DYNAMIC CLOSE CURSOR	37
COMMENT	COMMENT	–7

Table 86. SQL Statement Codes and Strings (continued)

Type of statement	Statement string	Statement code
COMMIT	COMMIT WORK	11
compound-statement	BEGIN END	12
CONNECT	CONNECT	13
CREATE ALIAS	CREATE ALIAS	–8
CREATE FUNCTION	CREATE ROUTINE	14
CREATE INDEX	CREATE INDEX	–14
CREATE PROCEDURE	CREATE ROUTINE	14
CREATE SCHEMA	CREATE SCHEMA	64
CREATE SEQUENCE	CREATE SEQUENCE	133
CREATE TABLE	CREATE TABLE	77
CREATE TRIGGER	CREATE TRIGGER	80
CREATE TYPE	CREATE TYPE	83
CREATE VARIABLE	CREATE VARIABLE	–83
CREATE VIEW	CREATE VIEW	84
DEALLOCATE DESCRIPTOR	DEALLOCATE DESCRIPTOR	15
DECLARE GLOBAL TEMPORARY TABLE	DECLARE GLOBAL TEMPORARY TABLE	–21
DELETE Positioned (static SQL)	DELETE CURSOR	18
DELETE Positioned (dynamic SQL)	DYNAMIC DELETE CURSOR	38
DELETE Searched	DELETE WHERE	19
DESCRIBE	DESCRIBE	20
DESCRIBE CURSOR	DESCRIBE CURSOR RESULT SET	–72
DESCRIBE PROCEDURE	DESCRIBE PROCEDURE	–23
DESCRIBE TABLE	DESCRIBE TABLE	–24
DISCONNECT	DISCONNECT	22
DROP ALIAS	DROP ALIAS	–25
DROP FUNCTION	DROP ROUTINE	30
DROP INDEX	DROP INDEX	–30
DROP PACKAGE	DROP PACKAGE	–32
DROP PROCEDURE	DROP ROUTINE	30
DROP SCHEMA	DROP SCHEMA	31
DROP SEQUENCE	DROP SEQUENCE	135
DROP TABLE	DROP TABLE	32
DROP TRIGGER	DROP TRIGGER	34
DROP TYPE	DROP TYPE	35
DROP VARIABLE	DROP VARIABLE	–84
DROP XSROBJECT	DROP XSROBJECT	–95
DROP VIEW	DROP VIEW	36
EXECUTE	EXECUTE	44

Table 86. SQL Statement Codes and Strings (continued)

Type of statement	Statement string	Statement code
EXECUTE IMMEDIATE	EXECUTE IMMEDIATE	43
FETCH (static SQL)	FETCH	45
FETCH (dynamic SQL)	DYNAMIC FETCH	39
FOR	FOR	46
FREE LOCATOR	FREE LOCATOR	98
GET DESCRIPTOR	GET DESCRIPTOR	47
GOTO	GOTO	–37
GRANT (any type)	GRANT	48
HOLD LOCATOR	HOLD LOCATOR	99
IF	IF	88
INSERT	INSERT	50
ITERATE	ITERATE	102
LABEL	LABEL	–39
LEAVE	LEAVE	89
LOCK TABLE	LOCK TABLE	–40
LOOP	LOOP	90
MERGE	MERGE	128
OPEN (static SQL)	OPEN	53
OPEN (dynamic SQL)	DYNAMIC OPEN	40
PREPARE	PREPARE	56
Prepared DELETE Positioned (dynamic SQL)	PREPARABLE DYNAMIC DELETE CURSOR	54
Prepared UPDATE Positioned (dynamic SQL)	PREPARABLE DYNAMIC UPDATE CURSOR	55
REFRESH TABLE	REFRESH TABLE	–41
RELEASE (connection)	RELEASE CONNECTION	–42
RELEASE SAVEPOINT	RELEASE SAVEPOINT	57
RENAME INDEX	RENAME INDEX	–43
RENAME TABLE	RENAME TABLE	–44
REPEAT	REPEAT	95
RESIGNAL	RESIGNAL	91
RETURN	RETURN	58
REVOKE (any type)	REVOKE	59
ROLLBACK	ROLLBACK WORK	62
SAVEPOINT	SAVEPOINT	63
SELECT INTO	SELECT	65
select-statement (dynamic SQL)	SELECT CURSOR	85
SET CONNECTION	SET CONNECTION	67
SET CURRENT DEBUG MODE	SET CURRENT DEBUG MODE	–75

Table 86. SQL Statement Codes and Strings (continued)

Type of statement	Statement string	Statement code
SET CURRENT DECFLOAT ROUNDING MODE	SET CURRENT DECFLOAT ROUNDING MODE	–82
SET CURRENT DEGREE	SET CURRENT DEGREE	–47
SET CURRENT IMPLICIT XMLPARSE OPTION	SET CURRENT IMPLICIT XMLPARSE OPT	–90
SET DESCRIPTOR	SET DESCRIPTOR	70
SET ENCRYPTION PASSWORD	SET ENCRYPTION PASSWORD	–48
SET PATH	SET PATH	69
SET RESULT SETS	SET RESULT SETS	–64
SET SCHEMA	SET SCHEMA	74
SET SESSION AUTHORIZATION	SET SESSION AUTHORIZATION	76
SET TRANSACTION	SET TRANSACTION	75
SET transition-variable	ASSIGNMENT	5
SET variable	ASSIGNMENT	5
SIGNAL	SIGNAL	92
UPDATE Positioned (static SQL)	UPDATE CURSOR	81
UPDATE Positioned (dynamic SQL)	DYNAMIC UPDATE CURSOR	42
UPDATE Searched	UPDATE WHERE	82
VALUES	STANDALONE FULLSELECT	–69
VALUES INTO	VALUES INTO	–66
WHILE	WHILE	97
Unrecognized statement	a zero length string	0

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword EXCEPTION can be used as a synonym for CONDITION.
- The keyword RETURN_STATUS can be used as a synonym for DB2_RETURN_STATUS.

Example

In an SQL procedure, execute a GET DIAGNOSTICS statement to determine how many rows were updated.

```
CREATE PROCEDURE sqlprocg (IN deptnbr VARCHAR(3))
  LANGUAGE SQL
  BEGIN
    DECLARE SQLSTATE CHAR(5);
    DECLARE rcount INTEGER;
    UPDATE CORPDATA.PROJECT
      SET PRSTAFF = PRSTAFF + 1.5
      WHERE DEPTNO = deptnbr;
    GET DIAGNOSTICS rcount = ROW_COUNT;
    /* At this point, rcount contains the number of rows that were updated. */
  END
```

Within an SQL procedure, handle the returned status value from the invocation of a stored procedure called TRYIT. TRYIT could use the RETURN statement to explicitly return a status value or a status value could be implicitly returned by the database manager. If the procedure is successful, it returns a value of zero.

```
CREATE PROCEDURE TESTIT ()
  LANGUAGE SQL
  A1: BEGIN
      DECLARE RETVAL INTEGER DEFAULT 0;
      ...
      CALL TRYIT
      GET DIAGNOSTICS RETVAL = RETURN_STATUS;
      IF RETVAL <> 0 THEN
         ...
         LEAVE A1;
      ELSE
         ...
      END IF;
  END A1
```

In an SQL procedure, execute a GET DIAGNOSTICS statement to retrieve the message text for an error.

```
CREATE PROCEDURE divide2 ( IN  numerator INTEGER,
                           IN  denominator INTEGER,
                           OUT divide_result INTEGER,
                           OUT divide_error VARCHAR(70) )
LANGUAGE SQL
  BEGIN
     DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        GET DIAGNOSTICS CONDITION 1
          divide_error = MESSAGE_TEXT;
     SET divide_result = numerator / denominator;
  END;
```

GRANT (Function or Procedure Privileges)

This form of the GRANT statement grants privileges on a function or procedure.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

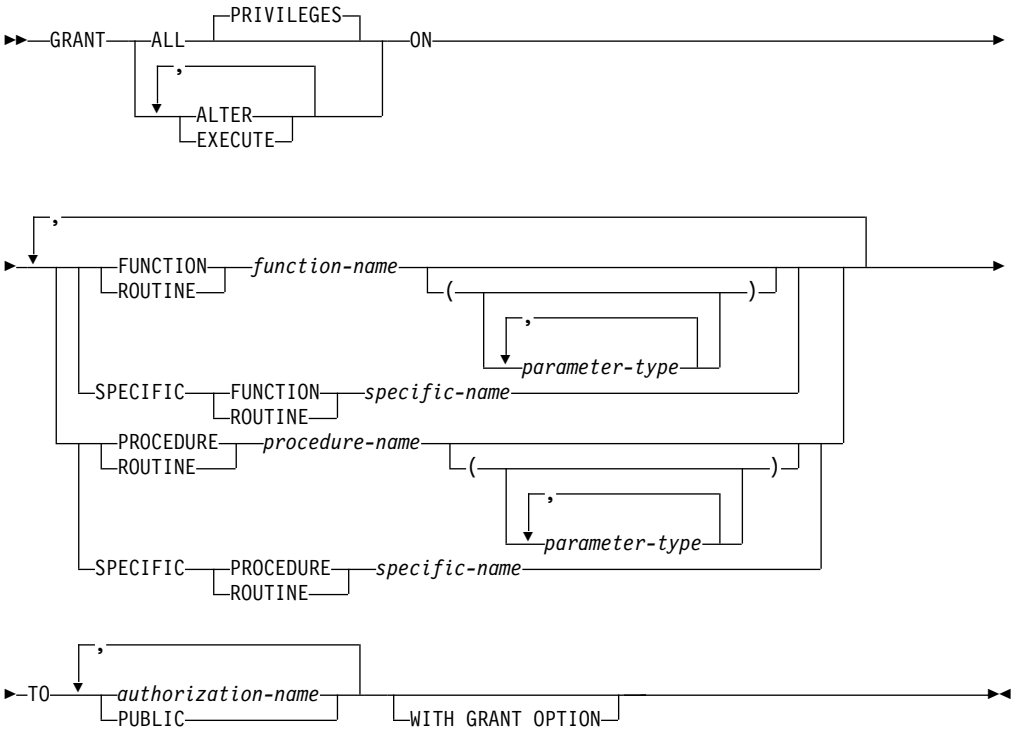
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each function or procedure identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the function or procedure
  - The system authority *EXECUTE on the library (or directory if this is a Java routine) containing the function or procedure
- Administrative authority

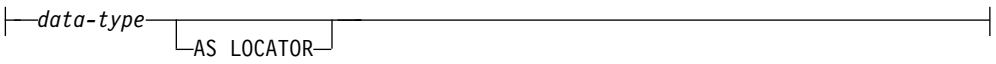
If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the function or procedure
- Administrative authority

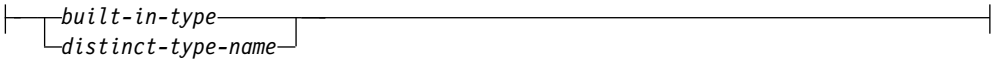
Syntax



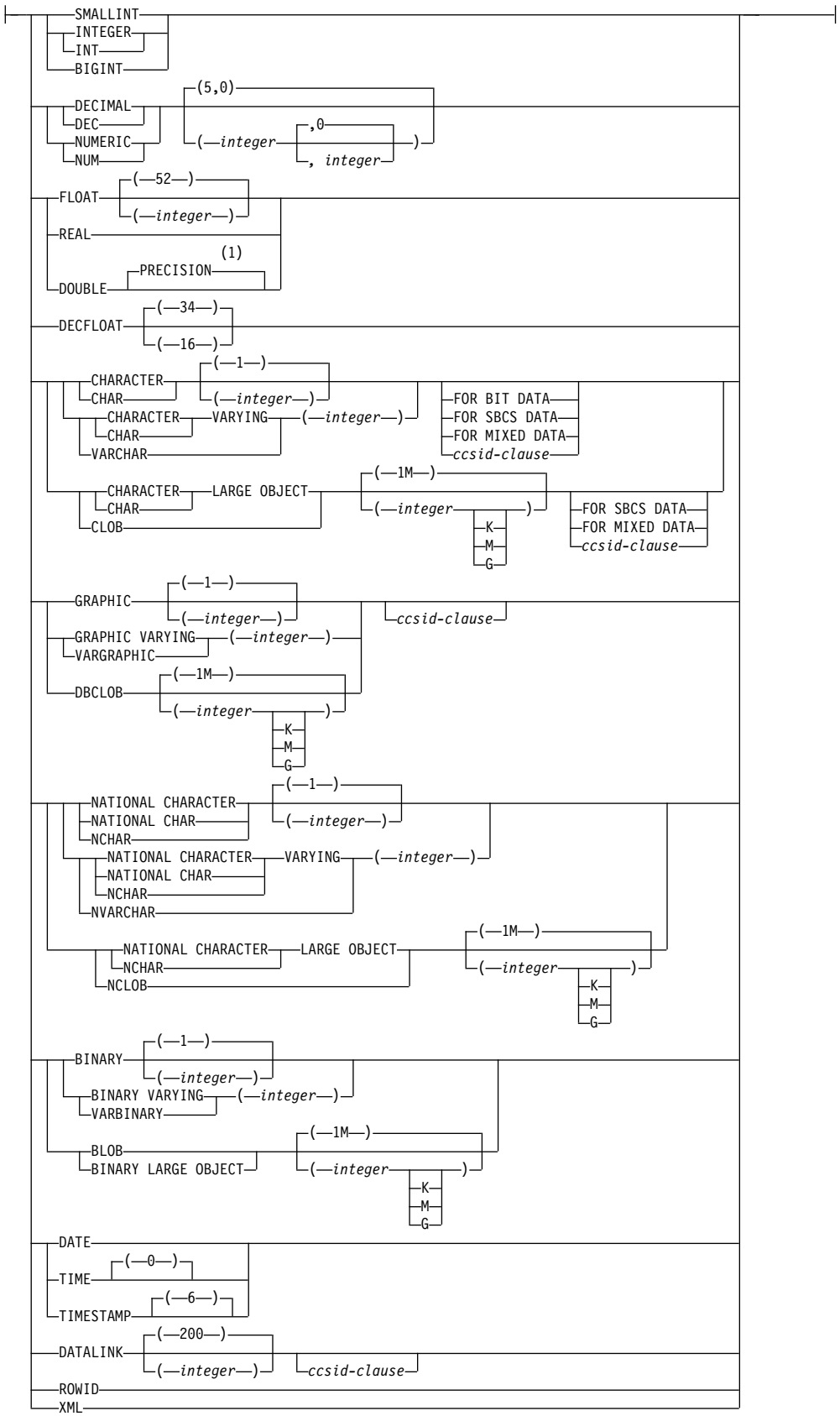
parameter-type:



data-type:



built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(integer)
,0
, integer
FLOAT
(52)
(integer)
REAL
DOUBLE
PRECISION
(1)
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
FOR BIT DATA
FOR SBCS DATA
FOR MIXED DATA
ccsid-clause
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
K
M
G
GRAPHIC
GRAPHIC VARYING
VARGRAPHIC
DBCLOB
NATIONAL CHARACTER
NATIONAL CHAR
NCHAR
NATIONAL CHARACTER VARYING
NATIONAL CHAR VARYING
NVARCHAR
NATIONAL CHARACTER LARGE OBJECT
NCHAR LARGE OBJECT
NCLOB
BINARY
BINARY VARYING
VARBINARY
BLOB
BINARY LARGE OBJECT
DATE
TIME
(0)
TIMESTAMP
(6)
DATALINK
(200)
ROWID
XML

ccsid-clause:

Notes:

1 The value that is specified for precision does not have to match the value that was specified when the function was created because matching is based on data type (REAL or DOUBLE).

Description

ALL or ALL PRIVILEGES
Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified functions or procedures. Note that granting ALL PRIVILEGES on a function or procedure is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

ALTER
Grants the privilege to use the ALTER FUNCTION, ALTER PROCEDURE, or COMMENT statement.

EXECUTE
Grants the privilege to execute the function or procedure.

FUNCTION or SPECIFIC FUNCTION
Identifies the function on which the privilege is granted. The function must exist at the current server and it must be a user-defined function, but not a function that was implicitly generated with the creation of a distinct type. The function can be identified by its name, function signature, or specific name.

FUNCTION *function-name*
Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*
Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which the privilege is to be granted. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name
Identifies the name of the function.

(parameter-type, ...)
Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE or SPECIFIC PROCEDURE
Identifies the procedure on which the privilege is granted. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*
Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be granted. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name
Identifies the name of the procedure.

(parameter-type, ...)
Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

TO Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 18.

WITH GRANT OPTION
Allows the specified *authorization-names* to grant privileges on the functions or procedures specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant privileges on the functions or procedures specified in the ON clause to

another user unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Notes

Corresponding System Authorities: Privileges granted to either an SQL or external function or procedure are granted to its associated program (*PGM) or service program (*SRVPGM) object. Privileges granted to a Java external function or procedure are granted to the associated class file or jar file. If the associated program, service program, class file, or jar file is not found when the grant is executed, an error is returned.

GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

Table 87. Privileges Granted to or Revoked from Non-Java Functions or Procedures

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Function or Procedure
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *EXECUTE *OBJMGT (Revoke only)
ALTER	*OBJALTER
EXECUTE	*EXECUTE *OBJOPR
WITH GRANT OPTION	*OBJMGT

Table 88. Privileges Granted to or Revoked from Java Functions or Procedures

SQL Privilege	Corresponding Data Authorities when Granting to or Revoking from a Java Function or Procedure	Corresponding Object Authorities when Granting to or Revoking from a Java Function or Procedure
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*RWX	*OBJEXIST *OBJALTER *OBJMGT (Revoke only)
ALTER	*R	*OBJALTER
EXECUTE	*RX	*EXECUTE
WITH GRANT OPTION	*RWX	*OBJMGT

Corresponding System Authorities When Checking Privileges to a Function or Procedure: The following table describes the system authorities that correspond to the SQL privileges when checking privileges to a function or procedure. The left column lists the SQL privilege. The right column lists the equivalent system authorities.

Table 89. Corresponding System Authorities When Checking Privileges to a Non-Java Function or Procedure

SQL Privilege	Corresponding System Authorities
ALTER	*OBJALTER
EXECUTE	*EXECUTE and *OBJOPR

Table 90. Corresponding System Authorities When Checking Privileges to a Java Function or Procedure

SQL Privilege	Corresponding Data Authorities when Checking Privileges to a Java Function or Procedure	Corresponding Object Authorities when Checking Privileges to a Java Function or Procedure
ALTER	*R	*OBJALTER
EXECUTE	*RX	*EXECUTE

Built-in functions: Privileges cannot be granted on built-in functions.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword RUN can be used as a synonym for EXECUTE.

Example

Grant the EXECUTE privilege on procedure PROCA to PUBLIC.

```
GRANT EXECUTE
  ON PROCEDURE PROCA
  TO PUBLIC
```

GRANT (Global Variable Privileges)

This form of the GRANT statement grants privileges on a global variable.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

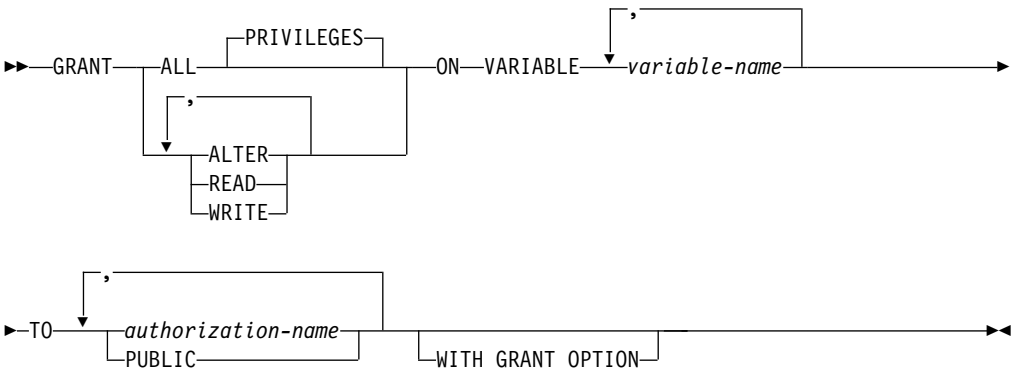
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each global variable identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the global variable
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the global variable
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified global variables. Note that granting ALL PRIVILEGES on a global variable is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

ALTER

Grants the privilege to use the COMMENT and LABEL statements on a global variables.

READ
Grants the privilege to read the value of a global variable.

WRITE
Grants the privilege to assign a value to a global variable.

ON VARIABLE *variable-name*
Identifies the global variables on which the privilege is granted. The *variable-name* must identify a global variable that exists at the current server.

TO Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 18.

WITH GRANT OPTION
Allows the specified *authorization-names* to grant privileges on the global variables specified in the ON clause to other users.

Notes

Corresponding System Authorities: GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

Table 91. Privileges Granted to or Revoked from Global Variables

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Global Variable
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *EXECUTE *UPD *READ *OBJMGT (Revoke only)
ALTER	*OBJALTER
READ	*OBJOPR *EXECUTE *READ
WRITE	*OBJOPR *EXECUTE *UPD
WITH GRANT OPTION	*OBJMGT

Corresponding System Authorities When Checking Privileges to a Global Variable: The following table describes the system authorities that correspond to the SQL privileges when checking privileges to a global variable. The left column lists the SQL privilege. The right column lists the equivalent system authorities.

Table 92. Corresponding System Authorities When Checking Privileges to a Global Variable

SQL Privilege	Corresponding System Authorities
ALTER	*OBJALTER
READ	*OBJOPR and *EXECUTE and *READ
WRITE	*OBJOPR and *EXECUTE and *UPD

Example

Grant the READ and WRITE privileges on global variable MYSCHEMA.MYJOB_PRINTER to user ZUBIRI.

```
GRANT READ, WRITE
  ON VARIABLE MYSCHEMA.MYJOB_PRINTEER
  TO ZUBIRI
```

GRANT (Package Privileges)

This form of the GRANT statement grants privileges on a package.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

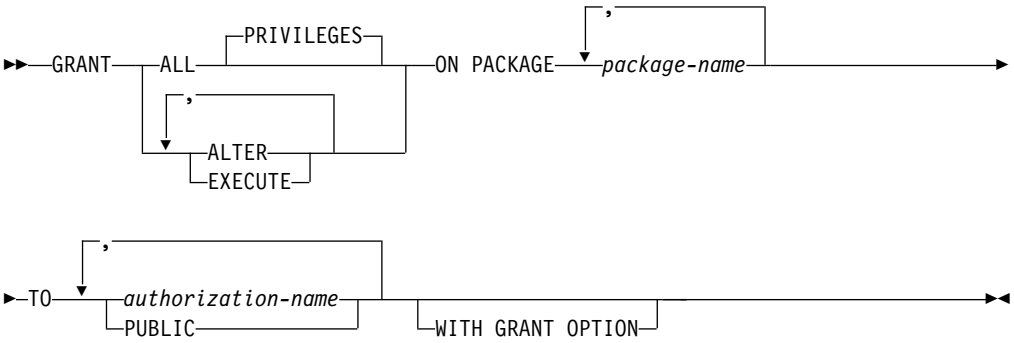
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each package identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the package
  - The system authority *EXECUTE on the library containing the package
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the package
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified packages. Note that granting ALL PRIVILEGES on a package is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

ALTER
Grants the privilege to use the COMMENT and LABEL statements.

EXECUTE
Grants the privilege to execute statements in a package.

ON PACKAGE *package-name*
Identifies the packages on which you are granting the privilege. The *package-name* must identify a package that exists at the current server.

TO Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 18.

WITH GRANT OPTION
Allows the specified *authorization-names* to grant privileges on the packages specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant privileges on the packages specified in the ON clause to another user unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Notes

Corresponding System Authorities: GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

Table 93. Privileges Granted to or Revoked from Packages

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Package
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *EXECUTE *OBJMGT (Revoke only)
ALTER	*OBJALTER
EXECUTE	*EXECUTE *OBJOPR
WITH GRANT OPTION	*OBJMGT

Corresponding System Authorities When Checking Privileges to a Package: The following table describes the system authorities that correspond to the SQL privileges when checking privileges to a package. The left column lists the SQL privilege. The right column lists the equivalent system authorities.

Table 94. Corresponding System Authorities When Checking Privileges to a Package

SQL Privilege	Corresponding System Authorities When Checking Privileges to a Package
ALTER	*OBJALTER
EXECUTE	*EXECUTE and *OBJOPR

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword RUN can be used as a synonym for EXECUTE.
- The keyword PROGRAM can be used as a synonym for PACKAGE.

Example

Grant the EXECUTE privilege on package PKGA to PUBLIC.

```
GRANT EXECUTE
  ON PACKAGE PKGA
  TO PUBLIC
```

GRANT (Sequence Privileges)

This form of the GRANT statement grants privileges on a sequence.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

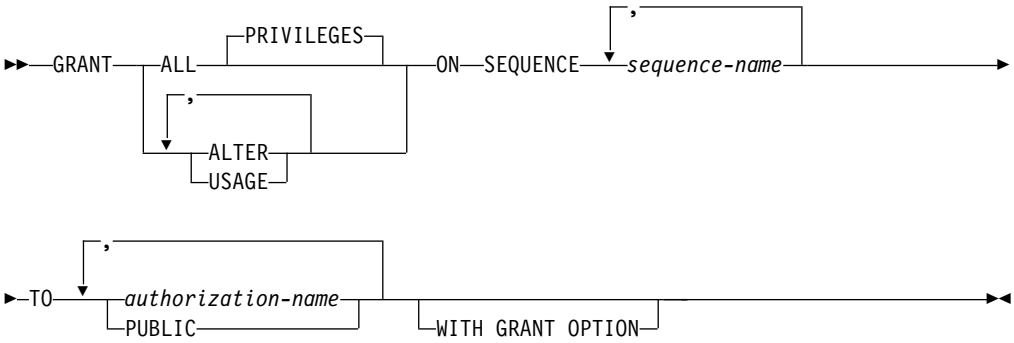
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each sequence identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the sequence
  - The system authority *EXECUTE on the library containing the sequence
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the sequence
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified sequences. Note that granting ALL PRIVILEGES on a sequence is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

ALTER

Grants the privilege to use the ALTER SEQUENCE, COMMENT, and LABEL statements on a sequence.

USAGE
Grants the privilege to use the sequence in NEXT VALUE or PREVIOUS VALUE expressions.

ON SEQUENCE *sequence-name*
Identifies the sequences on which the privilege is granted. The *sequence-name* must identify a sequence that exists at the current server.

TO Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 18.

WITH GRANT OPTION
Allows the specified *authorization-names* to grant privileges on the sequences specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant the USAGE privilege to others unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Notes

Corresponding System Authorities: GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

Table 95. Privileges Granted to or Revoked from Sequences

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Sequence
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *EXECUTE *READ *ADD *DLT *UPD *OBJMGT (Revoke only)
ALTER	*OBJALTER
USAGE	*OBJOPR *EXECUTE *READ *ADD *DLT *UPD
WITH GRANT OPTION	*OBJMGT

Corresponding System Authorities When Checking Privileges to a Sequence: The following table describes the system authorities that correspond to the SQL

privileges when checking privileges to a sequence. The left column lists the SQL privilege. The right column lists the equivalent system authorities.

Table 96. Corresponding System Authorities When Checking Privileges to a Sequence

SQL Privilege	Corresponding System Authorities
ALTER	*OBJALTER
USAGE	*OBJOPR and *EXECUTE and *READ and *ADD and *DLT and *UPD

Example

Grant any user the USAGE privilege on a sequence called ORG_SEQ.

```
GRANT USAGE
  ON SEQUENCE ORG_SEQ
  TO PUBLIC
```

GRANT (Table or View Privileges)

This form of the GRANT statement grants privileges on tables or views.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

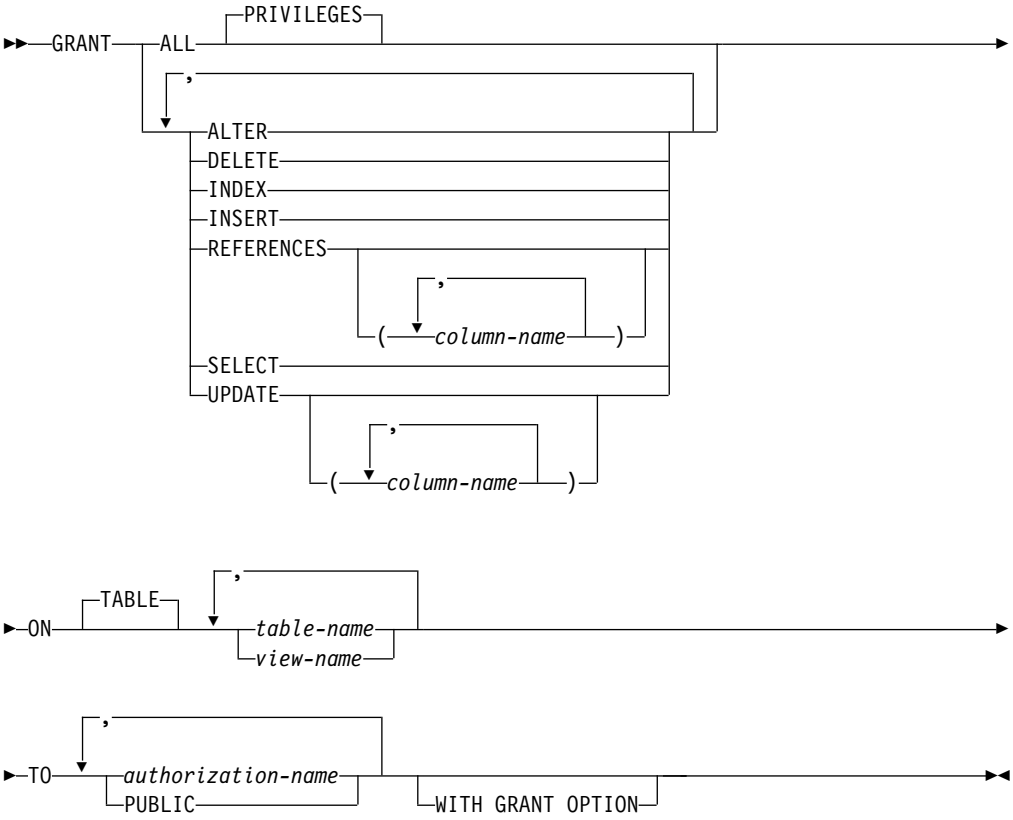
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the table or view
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the table
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified tables or views.

Note that granting ALL PRIVILEGES on a table or view is not the same as granting the system authority of *ALL.

ALTER
Grants the privilege to alter the specified table or create or drop a trigger on the specified table. Grants the privilege to use the COMMENT and LABEL statements on tables and views.

DELETE
Grants the privilege to delete rows from the specified table or view. If a view is specified, it must be a deletable view.

INDEX
Grants the privilege to create an index on the specified table. This privilege cannot be granted on a view.

INSERT
Grants the privilege to insert rows into the specified table or view. If a view is specified, it must be an insertable view.

REFERENCES
Grants the privilege to add a referential constraint in which each specified table is a parent. If a list of columns is not specified or if REFERENCES is granted to all columns of the table or view via the specification of ALL PRIVILEGES, the grantee(s) can add referential constraints using all columns of each table specified in the ON clause as a parent key, even those added later via the ALTER TABLE statement. This privilege can be granted on a view, but the privilege is not used for a view.

REFERENCES (*column-name*,...)
Grants the privilege to add a referential constraint in which each specified table is a parent using only those columns specified in the column list as a parent key. Each *column-name* must be an unqualified name that identifies a column of each table specified in the ON clause. This privilege can be granted on the columns of a view, but the privilege is not used for a view.

SELECT
Grants the privilege to create a view or read data from the specified table or view. For example, the SELECT privilege is required if a table or view is specified in a query.

UPDATE
Grants the privilege to update rows in the specified table or view. If a list of columns is not specified or if UPDATE is granted to all columns of the table or view via the specification of ALL PRIVILEGES, the grantee(s) can update all updatable columns on each table specified in the ON clause, even those added later via the ALTER TABLE statement. If a view is specified, it must be an updatable view.

UPDATE (*column-name*,...)
Grants the privilege to use the UPDATE statement to update only those columns that are identified in the column list. Each *column-name* must be an unqualified name that identifies a column of each table and view specified in

the ON clause. If a view is specified, it must be an updatable view and the specified columns must be updatable columns.

ON *table-name* **or** *view-name*,...
Identifies the tables or views on which the privileges are granted. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a declared temporary table.

TO Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 18.

WITH GRANT OPTION
Allows the specified *authorization-names* to grant privileges on the tables and views specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant privileges on the tables and views specified in the ON clause unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Notes

Corresponding system authorities: The GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges when granting to a table. The left column lists the SQL privilege. The right column lists the equivalent system authorities that are granted or revoked.

Table 97. Privileges Granted to or Revoked from Tables

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Table
ALL (GRANT or revoke of ALL only grants or revokes those privileges the authorization ID of the statement has)	*OBJALTER [103] *OBJMGT (Revoke only) *OBJOPR *OBJREF *ADD *DLT *READ *UPD
ALTER	*OBJALTER [104]
DELETE	*OBJOPR [105] *DLT
INDEX	*OBJALTER [104]
INSERT	*OBJOPR[105] *ADD
REFERENCES	*OBJREF [104]
SELECT	*OBJOPR[105] *READ
UPDATE	*OBJOPR[105] *UPD

Table 97. Privileges Granted to or Revoked from Tables (continued)

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Table
WITH GRANT OPTION	*OBJMGT

The following table describes the system authorities that correspond to the SQL privileges when granting to a view. The left column lists the SQL privilege. The middle column lists the equivalent system authorities that are granted to or revoked from the view itself. The right column lists the system authorities that are granted to all tables and views referenced in the view's definition, and if a view is referenced, all tables and views referenced in its definition, and so on. [106]

If a view references more than one table or view, the *DLT, *ADD, and *UPD system authorities are only granted to the first table or view in the fullselect of the view definition. The *READ system authority is granted to all tables and views referenced in the view definition.

If more than one system authority will be granted with an SQL privilege, and any one of the authorities cannot be granted, then a warning occurs and no authorities will be granted for that privilege. Unlike GRANT, REVOKE only revokes system authorities to the view. No system authorities are revoked from the referenced tables and views.

Table 98. Privileges Granted to or Revoked from Views

SQL Privilege	Corresponding System Authorities Granted to or Revoked from View	Corresponding System Authorities Granted to or Revoked from Referenced Tables and Views
ALL (GRANT or REVOKE of ALL only grants or revokes those privileges the authorization ID of the statement has)	*OBJALTER *OBJMGT (Revoke only) *OBJOPR *OBJREF *ADD *DLT *READ *UPD	*ADD *DLT *READ *UPD
ALTER	*OBJALTER [104]	None
DELETE	*OBJOPR[105] *DLT	*DLT
INDEX	Not Applicable	Not Applicable
INSERT	*OBJOPR[105] *ADD	*ADD
REFERENCES	*OBJREF [104]	None

103. The SQL INDEX and ALTER privilege correspond to the same system authority of *OBJALTER. Granting both INDEX and ALTER will not provide the user with any additional authorities.

104. If the WITH GRANT OPTION is given to a user, the user will also be able to perform the functions given by ALTER and REFERENCES authority.

105. *OBJOPR is only revoked when the last system privilege other than *OBJOPR is also revoked for the specified authorization ID or PUBLIC.

106. The specified rights are only granted to the tables and views referenced in the view definition if the user to whom the rights are being granted doesn't already have the rights from another authority source, for example public authority.

Table 98. Privileges Granted to or Revoked from Views (continued)

SQL Privilege	Corresponding System Authorities Granted to or Revoked from View	Corresponding System Authorities Granted to or Revoked from Referenced Tables and Views
SELECT	*OBJOPR[105] *READ	*READ
UPDATE	*OBJOPR[105] *UPD	*UPD
WITH GRANT OPTION	*OBJMGT	None

Corresponding system authorities when checking privileges to a table or view: The following table describes the system authorities that correspond to the SQL privileges when checking privileges to a table. The left column lists the SQL privilege. The right column lists the equivalent system authorities.

Table 99. Corresponding System Authorities when Checking Privileges to a Table

SQL Privilege	Corresponding System Authorities when Checking Privileges to a Table
ALTER	*OBJALTER or *OBJMGT
DELETE	*OBJOPR and *DLT
INDEX	*OBJALTER or *OBJMGT
INSERT	*OBJOPR and *ADD
REFERENCES	*OBJREF or *OBJMGT
SELECT	*OBJOPR and *READ
UPDATE	*OBJOPR and *UPD

The following table describes the system authorities that correspond to the SQL privileges when checking privileges to a view. The left column lists the SQL privilege. The middle column lists the equivalent system authorities that are checked on the view itself. The right column lists the system authorities that are checked on all tables and views referenced in the view's definition, and if a view is referenced, all tables and views referenced in its definition, and so on.

Table 100. Corresponding System Authorities when Checking Privileges to a View

SQL Privilege	Corresponding System Authorities to the View	Corresponding System Authorities to the Referenced Tables and Views
ALTER	*OBJALTER and *OBJMGT	None
DELETE[107]	*OBJOPR and *DLT	*DLT
INDEX	Not Applicable	Not Applicable
INSERT[108]	*OBJOPR and *ADD	*ADD
REFERENCES	*OBJREF or *OBJMGT	None
SELECT	*OBJOPR and *READ	*READ
UPDATE[109]	*OBJOPR and *UPD	*UPD

GRANT rules: The GRANT statement will grant only those privileges that the authorization ID of the statement is allowed to grant. If no privileges were granted, an error is returned.

Examples

Example 1: Grant all privileges on the table WESTERN_CR to PUBLIC.

```
GRANT ALL PRIVILEGES ON WESTERN_CR
  TO PUBLIC
```

Example 2: Grant the appropriate privileges on the CALENDAR table so that PHIL and CLAIRE can read it and insert new entries into it. Do not allow them to change or remove any existing entries.

```
GRANT SELECT, INSERT ON CALENDAR
  TO PHIL, CLAIRE
```

Example 3: Grant column privileges on TABLE1 and VIEW1 to FRED. Note that both columns specified in this GRANT statement must be found in both TABLE1 and VIEW1.

```
GRANT UPDATE(column_1, column_2)
  ON  TABLE1, VIEW1
  TO FRED WITH GRANT OPTION
```

107. When a view is created, the owner does not necessarily acquire the DELETE privilege on the view. The owner only acquires the DELETE privilege if the view allows deletes and the owner also has the DELETE privilege on the first table referenced in the subselect.

108. When a view is created, the owner does not necessarily acquire the INSERT privilege on the view. The owner only acquires the INSERT privilege if the view allows inserts and the owner also has the INSERT privilege on the first table referenced in the subselect.

109. When a view is created, the owner does not necessarily acquire the UPDATE privilege on the view. The owner only acquires the UPDATE privilege if the view allows updates and the owner also has the UPDATE privilege on the first table referenced in the subselect.

GRANT (Type Privileges)

This form of the GRANT statement grants privileges on a type.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

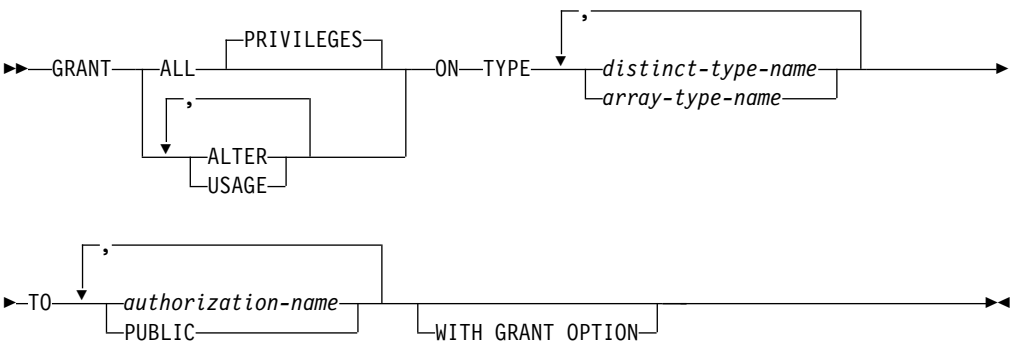
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type or array type identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the type
  - The system authority *EXECUTE on the library containing the type
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the type
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified distinct types or array types. Note that granting ALL PRIVILEGES on a type is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

ALTER
Grants the privilege to use the COMMENT and LABEL statement.

USAGE
Grants the privilege to use the type in tables, functions, procedures, or CAST expressions.

ON TYPE *distinct-type-name* **or** *array-type-name*
Identifies the distinct types or array types on which the privilege is granted. The *distinct-type-name* or *array-type-name* must identify a user-defined type that exists at the current server.

TO Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 18.

WITH GRANT OPTION
Allows the specified *authorization-names* to grant privileges on the types specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant the USAGE privilege to others unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Notes

Corresponding System Authorities: GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

Table 101. Privileges Granted to or Revoked from User-defined Types

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a User-defined Type
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *EXECUTE *OBJMGT (Revoke only)
ALTER	*OBJALTER
USAGE	*EXECUTE *OBJOPR
WITH GRANT OPTION	*OBJMGT

Corresponding System Authorities When Checking Privileges to a User-defined Type: The following table describes the system authorities that correspond to the SQL privileges when checking privileges to a type. The left column lists the SQL privilege. The right column lists the equivalent system authorities.

Table 102. Corresponding System Authorities When Checking Privileges to a User-defined Type

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a User-defined Type
ALTER	*OBJALTER

Table 102. Corresponding System Authorities When Checking Privileges to a User-defined Type (continued)

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a User-defined Type
USAGE	*EXECUTE and *OBJOPR

When USAGE privilege is required: USAGE privilege is required when a type is explicitly referenced in an SQL statement. For example, in a statement that contains a CAST specification or in a CREATE TABLE statement. The USAGE privilege is not required when a type is indirectly referenced. For example, when a view references a column of a table that has a distinct data type.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords DATA TYPE or DISTINCT TYPE can be used as a synonym for TYPE.

Example

Grant the USAGE privilege on distinct type SHOE_SIZE to user JONES. This GRANT statement does not give JONES the privilege to execute the cast functions that are associated with the distinct type SHOE_SIZE.

```
GRANT USAGE
  ON DISTINCT TYPE SHOE_SIZE
  TO JONES
```

GRANT (XML Schema Privileges)

This form of the GRANT statement grants privileges on an XSR object.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

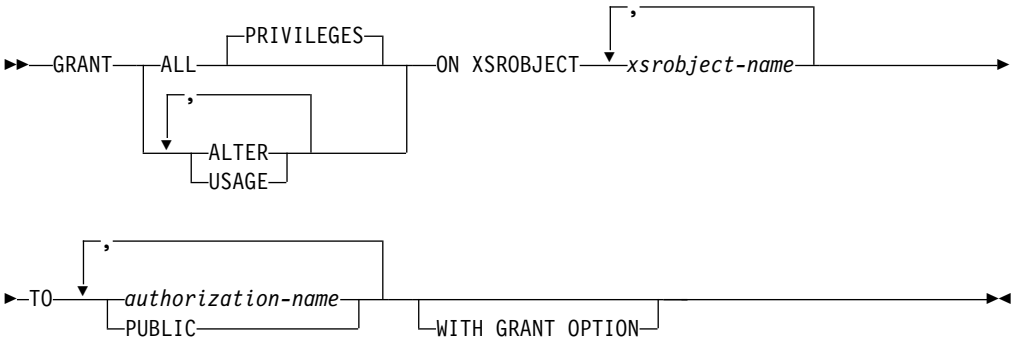
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each XSR object identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the XSR object
  - The system authority *EXECUTE on the library containing the XSR object
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the XSR object
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified XSR object. Note that granting ALL PRIVILEGES on an XSR object is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

ALTER
Grants the privilege to use the COMMENT and LABEL statements.

USAGE
Grants the privilege to use the XSR object for validation or decomposition.

ON XSROBJECT *xsrobject-name*
Identifies the XSR objects for which the privilege is granted. The *xsrobject-name* must identify an XSR object that exists at the current server.

TO Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 18.

WITH GRANT OPTION
Allows the specified *authorization-names* to grant privileges on the XSR objects specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant privileges on the XSR objects specified in the ON clause to another user unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Notes

GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

Table 103. Privileges Granted to or Revoked from XSR Objects

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from an XSR object
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *READ *EXECUTE *OBJMGT (Revoke only)
ALTER	*OBJALTER
USAGE	*OBJOPR *EXECUTE *READ
WITH GRANT OPTION	*OBJMGT

Corresponding System Authorities When Checking Privileges to an XSR Object: The following table describes the system authorities that correspond to the SQL privileges when checking privileges to an XSR object. The left column lists the SQL privilege. The right column lists the equivalent system authorities.

Table 104. Corresponding System Authorities When Checking Privileges to an XSR Object

SQL Privilege	Corresponding System Authorities
ALTER	*OBJALTER
USAGE	*OBJOPR and *EXECUTE and *READ

Example

Grant the USAGE privilege on XSR object XMLSCHEMA to PUBLIC.

```
GRANT USAGE
  ON XSROBJECT XMLSCHEMA
  TO PUBLIC
```

HOLD LOCATOR

The HOLD LOCATOR statement allows a LOB or XML locator variable to retain its association with a value beyond a unit of work.

Invocation

This statement can only be embedded in an application program. It cannot be issued interactively. It is an executable statement that can be dynamically prepared. However, the EXECUTE statement with the USING clause must be used to execute the prepared statement. HOLD LOCATOR cannot be used with the EXECUTE IMMEDIATE statement. It must not be specified in REXX.

Authorization

None required.

Syntax

Description

variable,...

Identifies a variable that must be declared in accordance with the rules for declaring variable locator variables. An indicator variable must not be specified. The locator variable type must be a binary large object locator, a character large object locator, a double-byte character large object locator, or an XML locator.

After the HOLD LOCATOR statement is executed, each locator variable in the *variable* list has the hold property.

The variable must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a CALL, FETCH, SELECT INTO, SET variable, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is raised.

If more than one variable is specified in the HOLD LOCATOR statement and an error occurs on one of the locators, no locators will be held.

Note

A LOB or XML locator variable that has the hold property is freed (has its association between it and its value removed) when:

- The SQL FREE LOCATOR statement is executed for the locator variable.
- The SQL ROLLBACK statement without a HOLD option is executed.
- The SQL session is terminated.

Example

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that locators have been established in a program to represent the

values represented by the columns. Give the CLOB locator variables LOCRES and LOCHIST, and the BLOB locator variable LOCPIC the hold property.

```
HOLD LOCATOR :LOCRES,:LOCHIST,:LOCPIC
```

INCLUDE

The INCLUDE statement inserts application code, including declarations and statements, into a source program.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

The authorization ID of the statement must have the system authorities *OBJOPR and *READ on the file that contains the member.

Syntax

Description

SQLCA
Indicates the description of an SQL communication area (SQLCA) is to be included. Include SQLCA must not be specified if the program includes a stand-alone SQLCODE or a stand-alone SQLSTATE. An SQLCA can be specified for C, C++, COBOL, and PL/I. If the SQLCA is not specified, the variable SQLCODE or SQLSTATE must appear in the program.

INCLUDE SQLCA must not be specified more than once in the same program. For more information, see "SQL diagnostic information" on page 708.

The SQLCA should not be specified for RPG programs. In an RPG program, the precompiler automatically includes the SQLCA.

For a description of the SQLCA, see Appendix C, "SQLCA (SQL communication area)," on page 1513.

SQLDA
Specifies the description of an SQL descriptor area (SQLDA) is to be included. INCLUDE SQLDA can be specified in C, C++, COBOL, PL/I, and ILE RPG.

For a description of the SQLDA, see Appendix D, "SQLDA (SQL descriptor area)," on page 1523.

name
Identifies a member or source stream file to be included in the source program.

If precompiling using the SRCFILE parameter, it identifies a member to be included from the file specified on the INCFILE parameter of the CRTSQL*xxx* command.

If precompiling using the SRCSTMF parameter, it identifies a file to be included using the path from the INCDIR parameter of the CRTSQL*xxx* command. No suffix will be appended to the name.

The source can contain any host language statements and any SQL statements other than an INCLUDE statement. In COBOL, INCLUDE *member-name* must not be specified in other than the DATA DIVISION or PROCEDURE DIVISION.

'string'
Identifies a file to be included using the path from the INCDIR parameter of the CRTSQL*xxx* command. The string will be handled as a normal SQL string literal; the source stream file rules for escaping characters will not be followed. No suffix will be appended to the string.

The source can contain any host language statements and any SQL statements other than an INCLUDE statement.

When your program is precompiled, the INCLUDE statement is replaced by source statements.

The INCLUDE statement must be specified at a point in your program where the resulting source statements are acceptable to the compiler.

Notes

CCSID considerations: If the CCSID of the file specified on the SRCFILE or SRCSTMF parameter is different from the CCSID of the source for the INCLUDE statement, the INCLUDE source is converted to the CCSID of the source file.

Example

Include an SQL descriptor area in a C program.

```
EXEC SQL INCLUDE SQLDA;

EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM TDEPT
    WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {
  EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;

/* Print results */

}

EXEC SQL CLOSE C1;
```

INSERT

The INSERT statement inserts rows into a table or view. Inserting a row into a view also inserts the row into the table on which the view is based if no INSTEAD OF INSERT trigger is defined on this view. If such a trigger is defined, the trigger will be activated instead.

There are three forms of this statement:

- The *INSERT using VALUES* form is used to insert one or more rows into the table or view using the values provided or referenced.
- The *INSERT using fullselect* form is used to insert one or more rows into the table or view using values from other tables or views.
- The *INSERT using n ROWS* form is used to insert multiple rows into the table or view using the values provided in a host-structure-array.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared with the exception of the n ROWS form, which must be a static statement embedded in an application program. The n ROWS form is not allowed in a REXX procedure.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

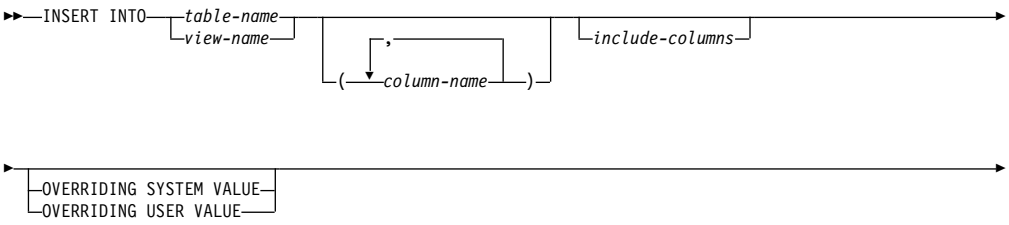
- For the table or view identified in the statement:
  - The INSERT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

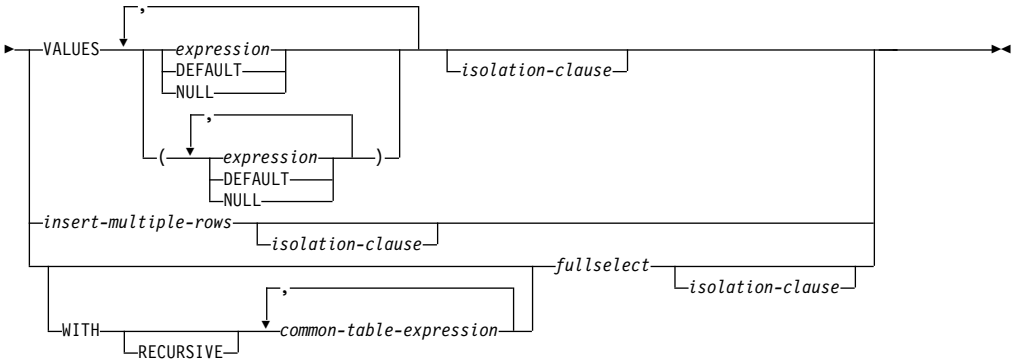
If a *fullselect* is specified, the privileges held by the authorization ID of the statement must also include one of the following:

- For each table or view identified in the *fullselect*:
  - The SELECT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

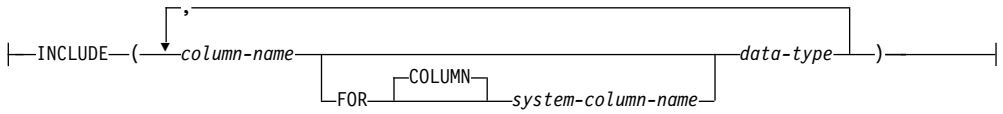
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

Syntax

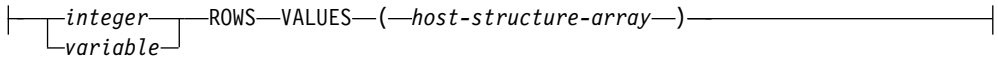




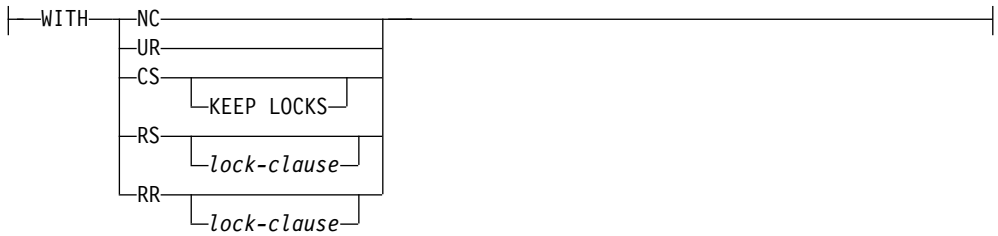
include-columns:



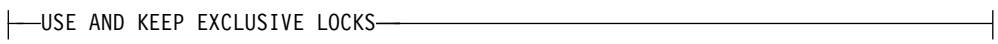
insert-multiple-rows:



isolation–clause:

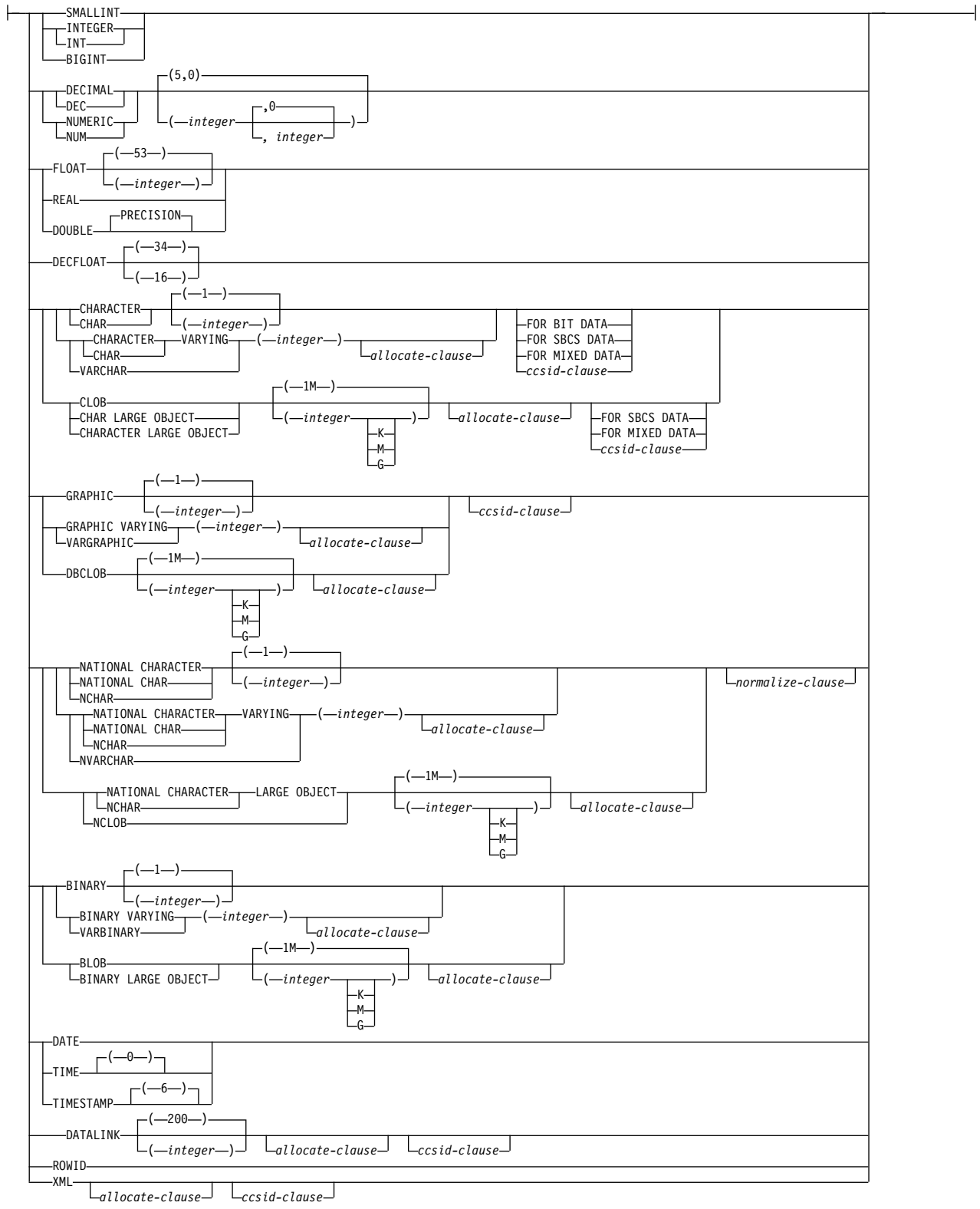


lock-clause:



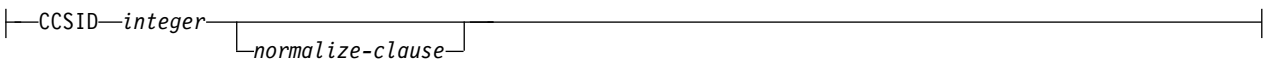
data-type:

built-in-type:



allocate-clause:

ccsid-clause:



normalize-clause:

Description

INTO *table-name* **or** *view-name*
Identifies the object of the insert operation. The name must identify a table or view that exists at the current server, but it must not identify a catalog table, a view of a catalog table, or a view that is not insertable. For an explanation of insertable views, see "CREATE VIEW" on page 1069.

(*column-name***,...)**
Specifies the columns for which insert values are provided. Each name must be a name that identifies a column of the table or view. The same column must not be identified more than once. If extended indicator variables are not enabled, a view column that is not updatable must not be identified. If extended indicator variables are not enabled and the object of the insert operation is a view with such columns, a list of column names must be specified and the list must not identify those columns. For an explanation of updatable columns in views, see "CREATE VIEW" on page 1069.

Omission of the column list is an implicit specification of a list in which every column of the table or view is identified in left-to-right order. Any columns defined with the hidden attribute are omitted. This list is established when the statement is prepared and, therefore, does not include columns that were added to a table after the statement was prepared.

If the INSERT statement is embedded in an application and the referenced table or view exists at create program time, the statement is prepared at create program time. Otherwise, the statement is prepared at the first successful execute of the INSERT statement.

include-columns
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the intermediate result table of the INSERT statement when it is nested in the FROM clause of a fullselect. The *include-columns* are appended to the end of the list of columns specified by *table-name* or *view-name*.

INCLUDE
Specifies a list of columns to be included in the intermediate result table of the INSERT statement. This clause can only be specified if the INSERT statement is nested in the FROM clause of a fullselect.

column-name
Specifies a column of the intermediate result table of the INSERT statement. The name cannot be the same as the name of another include column or a column in *table-name* or *view-name*.

FOR COLUMN *system-column-name*
Provides an IBM i name for the column. The name must not be the same as any *column-name* or *system-column-name* in the INCLUDE column list or in *table-name* or *view-name*.

data-type
Specifies the data type of the include column. For a description of *data-type*, see "CREATE TABLE" on page 982. If a DATALINK data-type is used, FILE LINK CONTROL is not allowed.

OVERRIDING SYSTEM VALUE or OVERRIDING USER VALUE
Specifies whether system generated values or user-specified values for a ROWID, identity, or row change timestamp column are used. If OVERRIDING SYSTEM VALUE is specified, the implicit or explicit list of columns for the INSERT statement must contain a column defined as GENERATED ALWAYS. If OVERRIDING USER VALUE is specified, the implicit or explicit list of columns for the INSERT statement must contain a column defined as either GENERATED ALWAYS or GENERATED BY DEFAULT.

OVERRIDING SYSTEM VALUE
Specifies that the value specified in the VALUES clause or produced by a fullselect for a column that is defined as GENERATED ALWAYS is used. A system-generated value is not inserted.

OVERRIDING USER VALUE
Specifies that the value specified in the VALUES clause or produced by a fullselect for a column that is defined as either GENERATED ALWAYS or GENERATED BY DEFAULT is ignored. Instead, a system-generated value is inserted, overriding the user-specified value.

If neither OVERRIDING SYSTEM VALUE nor OVERRIDING USER VALUE is specified:

- A value cannot be specified for a ROWID, identity, or row change timestamp column that is defined as GENERATED ALWAYS.
- A value can be specified for a ROWID, identity, or row change timestamp column that is defined as GENERATED BY DEFAULT. If a value is specified that value is assigned to the column. However, a value can be inserted into a ROWID column defined BY DEFAULT only if the specified value is a valid row ID value that was previously generated by DB2 for z/OS or DB2 for i. When a value is inserted into an identity or row change timestamp column defined BY DEFAULT, the database manager does not verify that the specified value is a unique value for the column unless the identity or row change timestamp column is the sole key in a unique constraint or unique index. Without a unique constraint or unique index, the database manager can guarantee unique values only among the set of system-generated values as long as NO CYCLE is in effect.

  If a value is not specified the database manager generates a new value.

VALUES
Specifies one or more new rows to be inserted.

Each variable in the clause must identify a host structure or variable that is declared in accordance with the rules for declaring host structures and variables. In the operational form of the statement, a reference to a host structure is replaced by a reference to each of its variables. For further information about variables and structures, see "References to host variables" on page 149 and "Host structures" on page 155.

The number of values for each row in the VALUES clause must equal the number of names in the implicit or explicit column list and the columns identified in the INCLUDE clause. The first value is inserted in the first column in the list, the second value in the second column, and so on.

expression
An *expression* of the type described in "Expressions" on page 165, that does not include an aggregate function or column name. If *expression* is a *variable*, the variable can identify a structure. If extended indicator variables are enabled and the expression is not a single variable, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used for that expression.

DEFAULT
Specifies that the default value is assigned to a column. The value that is inserted depends on how the column was defined, as follows:

- If the WITH DEFAULT clause is used, the default inserted is as defined for the column (see *default-clause* in *column-definition* in "CREATE TABLE" on page 982).
- If the WITH DEFAULT clause or the NOT NULL clause is not used, the value inserted is NULL.
- If the NOT NULL clause is used and the WITH DEFAULT clause is not used or DEFAULT NULL is used, the DEFAULT keyword cannot be specified for that column.
- If the column is a ROWID or identity column, the database manager will generate a new value.
- If the column is defined as a row change timestamp column, a new row change timestamp value is assigned to the column.

DEFAULT must be specified for a ROWID or an identity column that was defined as GENERATED ALWAYS unless OVERRIDING USER VALUE is specified to indicate that any user-specified value will be ignored and a unique system-generated value will be inserted.

NULL
Specifies the value for a column is the null value. NULL should only be specified for nullable columns.

WITH *common-table-expression*
Specifies a common table expression. For an explanation of common table expression, see "common-table-expression" on page 683.

fullselect
Specifies a set of new rows in the form of the result table of a fullselect. If the result table is empty, SQLSTATE is set to '02000'.

For an explanation of *fullselect*, see "fullselect" on page 676.

When the base object of the INSERT and a base object of any subselect in the *fullselect* are the same table, the select statement is completely evaluated before any rows are inserted.

The number of columns in the result table must equal the number of names implicitly or explicitly specified in the *column-name* list. The value of the first column of the result is inserted in the first column in the list, the second value in the second column, and so on.

isolation-clause
Specifies the isolation level to be used for this statement.

WITH

Introduces the isolation level, which may be one of:
- *RR* Repeatable read
- *RS* Read stability
- *CS* Cursor stability
- *UR* Uncommitted read
- *NC* No commit

If *isolation-clause* is not specified the default isolation is used. See "isolation-clause" on page 693 for a description of how the default is determined.

insert-multiple-rows

integer **or** *variable* **ROWS**
Specifies the number of rows to be inserted. If a *variable* is specified, it must be numeric with zero scale and cannot include an indicator variable.

VALUES (*host-structure-array***)**
Specifies a set of new rows in the form of an array of host structures. The *host-structure-array* must be declared in the program in accordance with the rules for declaring host structure arrays. A parameter marker may not be used in place of the *host-structure-array* name.

The number of variables in the host structure must equal the number of names in the implicit or explicit column list and the columns identified in the INCLUDE clause. The first host structure in the array corresponds to the first row, the second host structure in the array corresponds to the second row, and so on. In addition, the first variable in the host structure corresponds with the first column of the row, the second variable in the host structure corresponds with the second column of the row, and so on.

For an explanation of arrays of host structures see "Host structure arrays" on page 157.

insert-multiple-rows is not allowed if the current connection is to a non-DB2 for i remote server. *insert-multiple-rows* is not allowed in a data change reference in an RPG/400 or PL/I program.

INSERT Rules

Default Values: The value inserted in any column that is not in the column list is the default value of the column. Columns without a default value must be included in the column list. Similarly, if you insert into a view without an INSTEAD OF INSERT trigger, the default value is inserted into any column of the base table that is not included in the view. Hence, all columns of the base table that are not in the view must have a default value.

Assignment: Insert values are assigned to columns in accordance with the storage assignment rules described in Chapter 2, "Language elements," on page 49.

Validity: Insert operations must obey the following rules. If they do not, or if any other errors occur during the execution of the INSERT statement, no rows are inserted unless COMMIT(*NONE) was specified.
- *Unique constraints and unique indexes:* If the identified table, or the base table of the identified view, has one or more unique indexes or unique constraints, each

row inserted into the table must conform to the limitations imposed by those indexes and constraints (SQLSTATE 23505).

All uniqueness checks are effectively made at the end of the statement unless COMMIT(*NONE) was specified. In the case of a multiple-row INSERT statement, this would occur after all rows were inserted. If COMMIT(*NONE) is specified, checking is performed as each row is inserted.

- *Check constraints:* If the identified table, or the base table of the identified view, has one or more check constraints, each check constraint must be true or unknown for each row inserted into the table (SQLSTATE 23513).

  The check constraints are effectively checked at the end of the statement. In the case of a multiple-row INSERT statement, this would occur after all rows were inserted.
- *Views and the CHECK OPTION clause:* If a view is identified, the inserted rows must conform to any applicable CHECK OPTION clause (SQLSTATE 44000). For more information, see "CREATE VIEW" on page 1069.

Triggers: If the identified table or the base table of the identified view has an insert trigger, the trigger is activated. A trigger might cause other statements to be executed or raise error conditions based on the insert values. If the INSERT statement is used as a *data-change-table-reference*, an AFTER INSERT trigger that attempts to modify the inserted rows will cause an error.

Referential Integrity: Each nonnull insert value of a foreign key must equal some value of the parent key of the parent table in the relationship.

The referential constraints (other than a referential constraint with a RESTRICT delete rule) are effectively checked at the end of the statement. In the case of a multiple-row INSERT statement, this would occur after all rows were inserted and any associated triggers were activated.

If the INSERT statement is used as a *data-change-table-reference*, any referential constraint that attempts to modify the inserted rows will cause an error.

XML values: A value that is inserted into an XML column must be a well-formed XML document.

Extended indicator variable usage: If enabled, indicator variable values other than positive values and 0 (zero) through -7 must not be set. The DEFAULT and UNASSIGNED extended indicator variable values must not appear in contexts where they are not supported.

Extended indicator variables: In an INSERT statement, the extended indicator value of UNASSIGNED has the effect of setting the column to its default value.

If a target column is not updatable, it must be assigned the extended indicator variable value of UNASSIGNED, unless it is an identity column defined as GENERATED ALWAYS. If the target column is an identity column defined as GENERATED ALWAYS; then it must be assigned the DEFAULT keyword or the extended indicator variable values of DEFAULT or UNASSIGNED.

Extended indicator variables and insert triggers: No change in the activation of insert triggers results from the presence of extended indicator variables. If all columns in the implicit or explicit column list have been assigned to an extended

indicator variable-based value of UNASSIGNED, or DEFAULT, an insert where all columns have their respective DEFAULT values is attempted, and if successful, the insert trigger is activated.

Extended indicator variables and deferred error checks: When extended indicator variables are enabled, validation that would normally be done during statement preparation to recognize an insert into a non-updatable column is deferred until the statement is executed.

Notes

Insert operation errors: If an insert value violates any constraints, or if any other error occurs during the execution of an INSERT statement and COMMIT(*NONE) was not specified, all changes from this statement and any triggered SQL statements are rolled back. However, other changes in the unit of work made prior to the error are not rolled back. If COMMIT(*NONE) is specified, changes are not rolled back.

Number of rows inserted: After executing an INSERT statement, the ROW_COUNT statement information item in the SQL Diagnostics Area (or SQLERRD(3) of the SQLCA) is the number of rows that the database manager inserted. The ROW_COUNT item does not include the number of rows that were inserted as a result of a trigger.

For a description of ROW_COUNT, see "GET DIAGNOSTICS" on page 1194. For a description of the SQLCA, see Appendix C, "SQLCA (SQL communication area)," on page 1513.

Locking: If COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) is specified, one or more exclusive locks are acquired during the execution of a successful INSERT statement. Until the locks are released by a commit or rollback operation, an inserted row can only be accessed by:

- The application process that performed the insert
- Another application process using COMMIT(*NONE) or COMMIT(*CHG) through a read-only operation

The locks can prevent other application processes from performing operations on the table. For further information about locking, see the description of the "COMMIT" on page 824, "ROLLBACK" on page 1338, and "LOCK TABLE" on page 1272 statements. Also, see "Isolation level" on page 26 and Database Programming.

If the INSERT is used as a *data-change-table-reference* where FINAL TABLE is specified, locks are placed on inserted rows until the SELECT is complete. These locks may prevent indirect changes to the inserted rows from within the same job, such as an AFTER TRIGGER attempting to change an inserted row. These locks are acquired for all isolation levels, including COMMIT(*NONE).

A maximum of 500 000 000 rows can be inserted or changed in any single INSERT statement when COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) was specified. The number of rows changed includes any rows inserted, updated, or deleted under the same commitment definition as a result of a trigger.

Row change timestamp columns: A row change timestamp column that is defined as GENERATED ALWAYS should not be specified in the *column-list* unless the corresponding entry in the VALUES list is DEFAULT. The user can specify the OVERRIDING USER VALUE clause to indicate that any user-specified value will be ignored and the timestamp value associated with the INSERT will be inserted into this column.

REXX: Variables cannot be used in the INSERT statement within a REXX procedure. Instead, the INSERT must be the object of a PREPARE and EXECUTE using parameter markers.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword NONE can be used as a synonym for NC.
- The keyword CHG can be used as a synonym for UR.
- The keyword ALL can be used as a synonym for RS.

Examples

Example 1: Insert a new department with the following specifications into the DEPARTMENT table:

- Department number (DEPTNO) is 'E31'
- Department name (DEPTNAME) is 'ARCHITECTURE'
- Managed by (MGRNO) a person with number '00390'
- Reports to (ADMRDEPT) department 'E01'.

```
INSERT INTO DEPARTMENT
  VALUES ('E31', 'ARCHITECTURE', '00390', 'E01')
```

Example 2: Insert a new department into the DEPARTMENT table as in example 1, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
  VALUES ('E31', 'ARCHITECTURE', 'E01')
```

Example 3: Create a table MA_EMPPROJACT with the same columns as the EMPPROJACT table. Populate MA_EMPPROJACT with the rows from the EMPPROJACT table with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE TABLE MA_EMPPROJACT LIKE EMPPROJACT

INSERT INTO MA_EMPPROJACT
  SELECT * FROM EMPPROJACT
    WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 4: Use a Java program statement to add a skeleton project to the PROJECT table on the connection context 'ctx'. Obtain the project number (PROJNO), project name (PROJNAME), department number (DEPTNO), and responsible employee (RESPEMP) from host variables. Use the current date as the project start date (PRSTDATE). Assign a NULL value to the remaining columns in the table.

```
#sql [ctx] { INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
             VALUES (:PRJNO, :PRJNM, :DPTNO, :REMP, CURRENT DATE) };
```

Example 5: Insert two new departments using one statement into the DEPARTMENT table as in example 2, but do not assign a manager to the new departments.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
  VALUES ('B11', 'PURCHASING', 'B01'),
         ('E41', 'DATABASE ADMINISTRATION', 'E01')
```

Example 6: In a PL/I program, use a multiple-row INSERT to add 10 rows to table DEPARTMENT. The host structure array DEPT contains the data to be inserted.

```
DCL  1  DEPT(10),
        3  DEPT  CHAR(3),
        3  LASTNAME  CHAR(29)  VARYING,
        3  WORKDEPT  CHAR(6),
        3  JOB  CHAR(3);

EXEC SQL INSERT INTO DEPARTMENT 10 ROWS VALUES (:DEPT);
```

Example 7: Insert a new project into the EMPPROJACT table using the Read Uncommitted (UR, CHG) option:

```
INSERT INTO EMPPROJACT
  VALUES ('000140', 'PL2100', 30)
  WITH CHG
```

Example 8: Specify an INSERT statement as the *data-change-table-reference* within a SELECT statement. Define an extra include column whose values are specified in the VALUES clause, which is then used as an ordering column for the inserted rows.

```
SELECT inorder, ordernum
  FROM FINAL TABLE (INSERT INTO ORDERS (CUSTNO)
                                INCLUDE(INSERTNUM INTEGER)
                      VALUES (:cnum1, 1),
                             (:cnum2, 2)) InsertedOrders
  ORDER BY insertnum
```

LABEL

The LABEL statement adds or replaces labels in the catalog descriptions of various database objects.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

To label a table, view, alias, column, type, package, sequence, view, or XSR object, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the table, view, alias, type, package, sequence, variable, XSR object, function or procedure identified in the statement,
  - The ALTER privilege on the table, view, alias, type, package, sequence, variable, XSR object, function or procedure, and
  - The system authority *EXECUTE on the library containing the table, view, alias, type, package, sequence, variable, XSR object, function or procedure
- Administrative authority

To label a constraint or trigger, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the subject table of the constraint or trigger in the statement:
  - The ALTER privilege on the subject table, and
  - The system authority *EXECUTE on the library that contains the subject table
- Administrative authority

To label an index, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the index identified in the statement,
  - The system authority *OBJALTER on the index, and
  - The system authority *EXECUTE on the library containing the index.
- Administrative authority

To label a function, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSFUNCS and SYSROUTINES catalog view and table:
  - The UPDATE privilege on SYSROUTINES,
  - The system authority *OBJOPR on SYSFUNCS, and
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

To label a procedure, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSPROCS and SYSROUTINES catalog view and table:
  - The UPDATE privilege on SYSROUTINES,
  - The system authority *OBJOPR on SYSPROCS, and
  - The system authority *EXECUTE on library QSYS2

- Administrative authority

To label a sequence, the privileges held by the authorization ID of the statement must also include at least one of the following:

- *USE authority to the Change Data Area (CHGDTAARA) CL command
- Administrative authority

The authorization ID of the statement has the ALTER privilege on an alias when:

- It is the owner of the alias, or
- It has been granted the system authorities of either *OBJALTER or *OBJMGT to the alias

To label a variable, the privileges held by the authorization ID of the statement must also include at least one of the following:

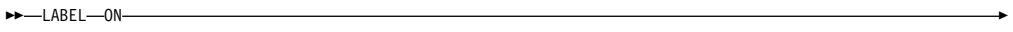
- For the SYSVARIABLES catalog table:
  - The UPDATE privilege on SYSVARIABLES,
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

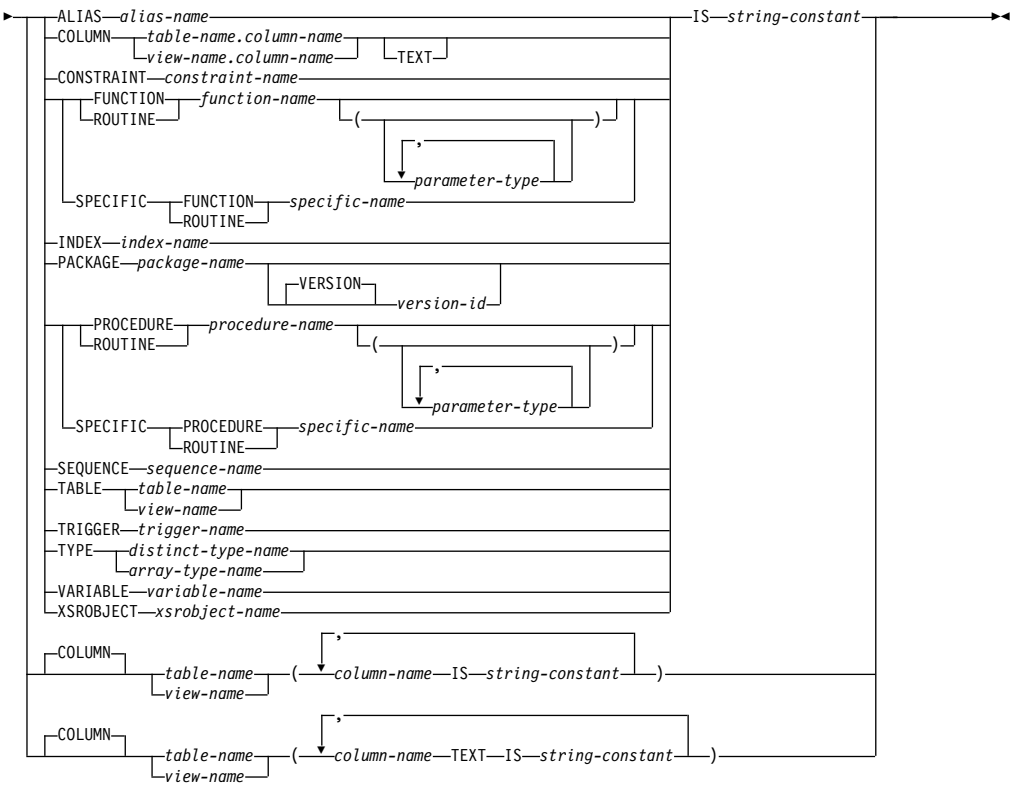
To label an XSR object, the privileges held by the authorization ID of the statement must also include at least one of the following:

- For the XSROBJECTS catalog table:
  - The UPDATE privilege on XSROBJECTS,
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

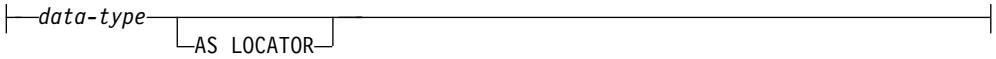
For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View, Corresponding System Authorities When Checking Privileges to a Sequence, and Corresponding System Authorities When Checking Privileges to a Package.

Syntax



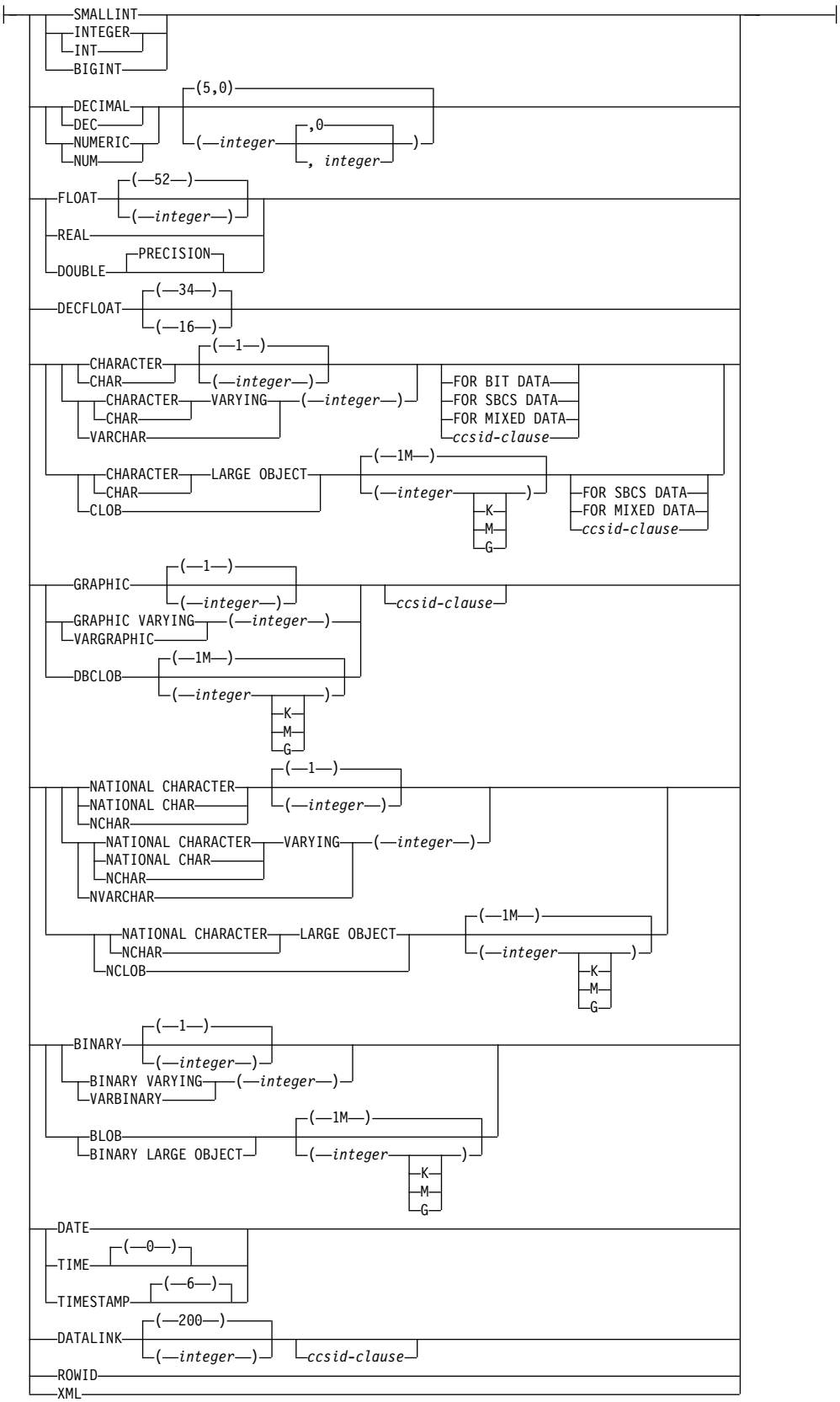


parameter-type:



data-type:

built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(integer
,0
, integer
)
FLOAT
(52)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
FOR BIT DATA
FOR SBCS DATA
FOR MIXED DATA
ccsid-clause
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
K
M
G
GRAPHIC
GRAPHIC VARYING
VARGRAPHIC
DBCLOB
NATIONAL CHARACTER
NATIONAL CHAR
NCHAR
NATIONAL CHARACTER VARYING
NATIONAL CHAR VARYING
NCHAR VARYING
NVARCHAR
NATIONAL CHARACTER LARGE OBJECT
NCHAR LARGE OBJECT
NCLOB
BINARY
BINARY VARYING
VARBINARY
BLOB
BINARY LARGE OBJECT
DATE
TIME
(0)
TIMESTAMP
(6)
DATALINK
(200)
ROWID
XML

ccsid-clause:

Description

ALIAS
Specifies that the label is for an alias. Labels on aliases are implemented as system object text.

alias-name
Identifies the alias to which the label applies. The name must identify an alias that exists at the current server.

COLUMN
Specifies that the label is for a column. Labels on columns are implemented as system column headings or column text. Column headings are used when displaying or printing query results.

table-name.column-name **or** *view-name.column-name*
Identifies the column to which the label applies. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a declared temporary table. The *column-name* must identify a column of that table or view.

TEXT
Specifies that IBM i column text is specified. If TEXT is omitted, a column heading is specified.

CONSTRAINT
Specifies that the label is for a constraint.

constraint-name
Identifies the constraint to which the label applies. The *constraint-name* must identify a constraint that exists at the current server.

FUNCTION or SPECIFIC FUNCTION
Identifies the function on which the label applies. The function must exist at the current server and it must be a user-defined function. The function can be identified by its name, function signature, or specific name.

FUNCTION *function-name*
Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*
Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which to label. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name
Identifies the name of the function.

(parameter-type, ...)
Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

INDEX
Specifies that the label is for an index. Labels on indexes are implemented as system object text.

index-name
Identifies the index to which the label applies. The name must identify an index that exists at the current server.

PACKAGE
Specifies that the label is for a package. Labels on packages are implemented as system object text.

package-name
Identifies the package to which the label applies. The name must identify a package that exists at the current server.

VERSION *version-id*
version-id is the version identifier that was assigned to the package when it was created. If *version-id* is not specified, a null string is used as the version identifier.

PROCEDURE or SPECIFIC PROCEDURE
Identifies the procedure to which the label applies. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*
Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be labeled on. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name
Identifies the name of the procedure.

(parameter-type, ...)
Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type

are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML. If AS LOCATOR is specified, FOR SBCS DATA or FOR MIXED DATA must not be specified.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

SEQUENCE
Specifies that the label is for a sequence. Labels on sequences are implemented as system object text.

sequence-name
Identifies the sequence on which you want to add a label. The *sequence-name* must identify a sequence that exists at the current server.

TABLE
Specifies that the label is for a table or a view. Labels on tables or views are implemented as system object text.

table-name **or** *view-name*
Identifies the table or view on which you want to add a label. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a declared temporary table.

TRIGGER
Specifies that the label is for a trigger.

trigger-name
Identifies the trigger on which you want to add a label. The *trigger-name* must identify a trigger that exists at the current server.

TYPE *distinct-type-name* **or** *array-type-name*
Identifies the distinct type or array type to which the label applies. The *distinct-type-name* or *array-type-name* must identify a type that exists at the current server.

VARIABLE *variable-name*
Identifies the variable to which the label applies. The *variable-name* must identify a variable that exists at the current server.

XSROBJECT *xsrobject-name*
Identifies the XSR object to which the label applies. The *xsrobject-name* must identify an XSR object that exists at the current server.

IS
Introduces the label you want to provide.

string-constant
Can be any SQL character-string constant of up to either 60 bytes in length

for column headings or 50 bytes in length for object text or column text. The constant may contain single-byte and double-byte characters.

The label for a column heading consists of three 20-byte segments. Interactive SQL, the Query for IBM i, IBM DB2 Query Manager and SQL Development Kit for i, and other products can display or print each 20-byte segment on a separate line. If the label for a column contains mixed data, each 20-byte segment must be a valid mixed data character string. The shift characters must be paired within each 20-byte segment.

Notes

Column headings: Column headings are used when displaying or printing query results. The first column heading is displayed or printed on the first line, the second column heading is displayed or printed on the second line, and the third column heading is displayed or printed on the third line. The column headings can be up to 60 bytes in length, where the first 20 bytes is the first column heading, the second 20 bytes is the second column heading, and the third 20 bytes is the third column heading. Blanks are trimmed from the end of each 20-byte column heading.

All 60 bytes of column heading information are available in the catalog view SYSCOLUMNS; however, only the first column heading is returned in an SQLDA on a DESCRIBE or DESCRIBE TABLE statement.

Column text is not returned on a DESCRIBE or DESCRIBE TABLE statement. When the database manager changes the column heading information in a record format description that is shared, the change is reflected in all files sharing the format description. To find out if a file shares a format with another file, use the RCDFMT parameter on the CL command, Display Database Relations (DSPDBR).

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword PROGRAM can be used as a synonym for PACKAGE.
- The keywords DATA TYPE or DISTINCT TYPE can be used as a synonym for TYPE.

Examples

Example 1: Enter a label on the DEPTNO column of table DEPARTMENT.

```
LABEL ON COLUMN DEPARTMENT.DEPTNO
  IS 'DEPARTMENT NUMBER'
```

Example 2: Enter a label on the DEPTNO column of table DEPARTMENT where the column heading is shown on two separate lines.

```
LABEL ON COLUMN DEPARTMENT.DEPTNO
  IS 'Department          Number'
```

Example 3: Enter a label on the PAYROLL package.

```
LABEL ON PACKAGE PAYROLL
  IS 'Payroll Package'
```

LOCK TABLE

The LOCK TABLE statement either prevents concurrent application processes from changing a table or prevents concurrent application processes from using a table.

Invocation

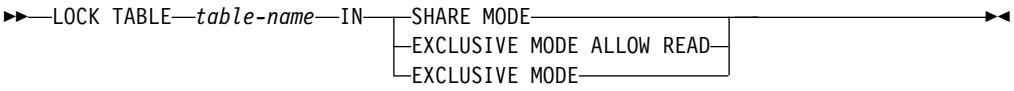
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table identified in the statement,
  - The system authority of *OBJOPR on the table, and
  - The system authority *EXECUTE on the library containing the table
- Administrative authority

Syntax



Description

table-name
Identifies the table to be locked. The table-name must identify a base table that exists at the current server, but must not identify a catalog table or a declared temporary table.

IN SHARE MODE
Prevents concurrent application processes from executing any but read-only operations on the table.

A shared lock (*SHRNUP) is acquired for the application process in which the statement is executed. Other application processes may also acquire a shared lock (*SHRNUP) and prevent this application process from executing any but read-only operations.

IN EXCLUSIVE MODE ALLOW READ
Prevents concurrent application processes from executing any but read-only operations on the table.

An exclusive allow read lock (*EXCLRD) is acquired for the application process in which the statement is executed. Other application processes may not acquire a shared lock (*SHRNUP) and cannot prevent this application process from executing updates, deletes, and inserts on the table.

IN EXCLUSIVE MODE
Prevents concurrent application processes from executing any operations at all on the table.

An exclusive lock (*EXCL) is acquired for the application process in which the statement is executed.

Notes

Locks obtained: Locking is used to prevent concurrent operations.

The lock is released:

- When the unit of work ends, unless the unit of work is ended by a COMMIT HOLD or ROLLBACK HOLD
- When the first SQL program in the program stack ends, unless CLOSQLCSR(*ENDJOB) or CLOSQLCSR(*ENDACTGRP) was specified on the CRTSQLxxx command
- When the activation group ends
- When the connection is changed using a CONNECT (Type 1) statement
- When the connection associated with the lock is disconnected using the DISCONNECT statement
- When the connection is in the release-pending state and a successful COMMIT occurs

You may also issue the Deallocate Object (DLCOBJ) command to unlock the table.

Lock wait time: Conflicting locks already held by other application processes will cause your application to wait up to the default wait time of the job.

Example

Obtain a lock on the DEPARTMENT table.

```
LOCK TABLE DEPARTMENT IN EXCLUSIVE MODE
```

MERGE

The MERGE statement updates a target (a table or view) using data from a source (result of a table reference). Rows in the target that match the input data may be updated or deleted as specified, and rows that do not exist in the target may be inserted as specified. Updating, deleting, or inserting a row in a view updates, deletes, or inserts the row into the tables on which the view is based if no INSTEAD OF trigger is defined on the view.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared. It is not allowed in a REXX procedure.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Administrative authority
- If an insert operation is specified:
  - The INSERT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- If a delete operation is specified:
  - The DELETE privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- If an update operation is specified:
  - The UPDATE privilege on the table or view or
  - The UPDATE privilege on each column to be updated; and
  - The system authority *EXECUTE on the library containing the table or view

If *search-condition*, *insert-operation*, or *assignment-clause* includes a fullselect, the privileges held by the authorization ID of the statement must also include at least one of the following:

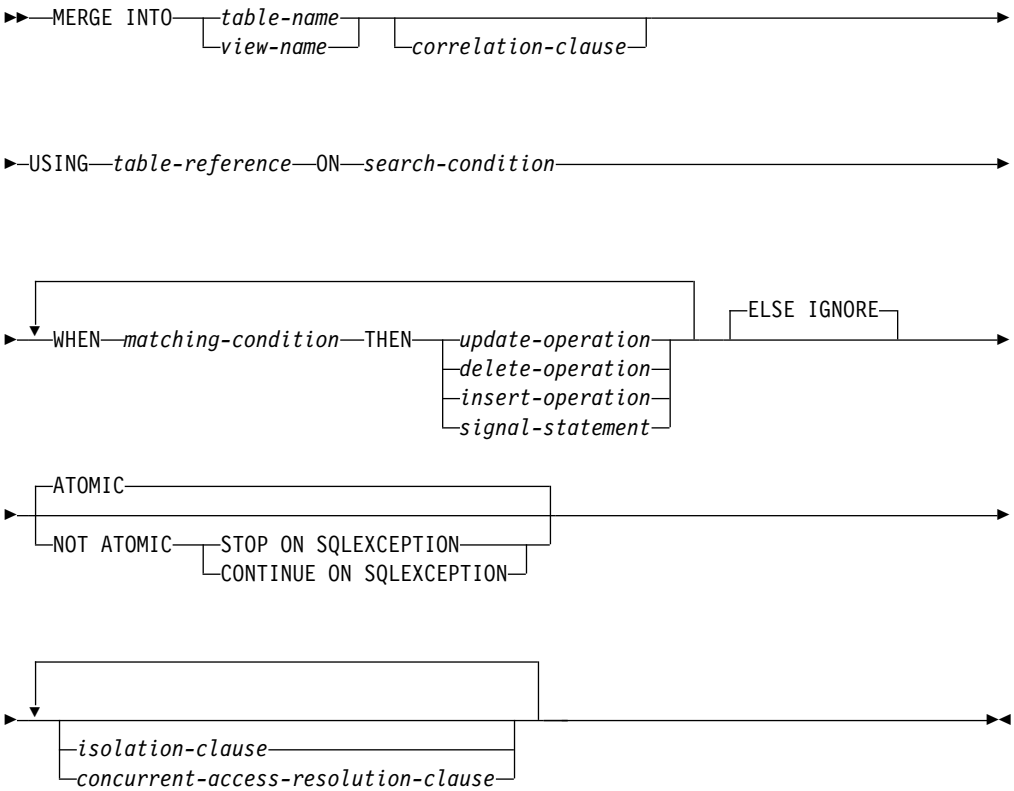
- For each table or view identified in the *fullselect*:
  - The SELECT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

If *table-reference* contains any query that references a column of a table or view, the privileges held by the authorization ID of the statement must also include at least one of the following:

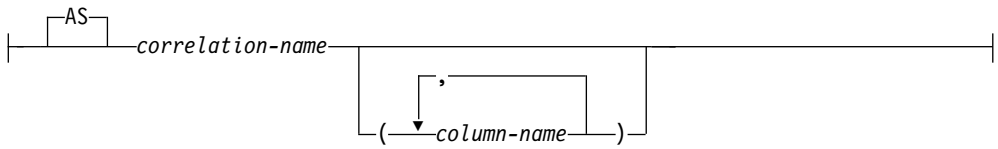
- For each table or view identified in the *fullselect*:
  - The SELECT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

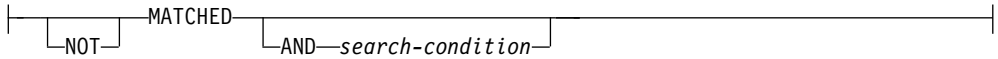
Syntax



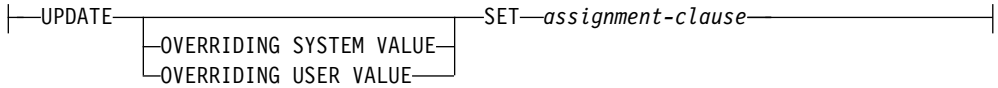
correlation-clause:



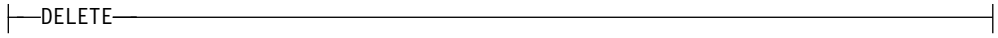
matching-condition:



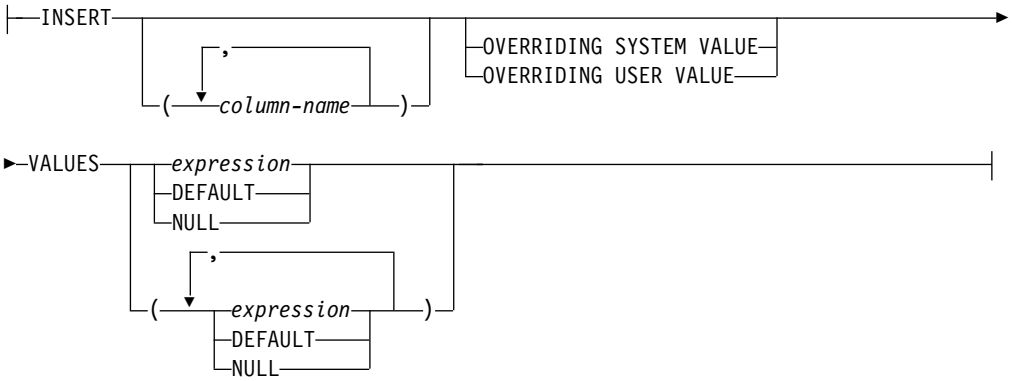
update-operation:



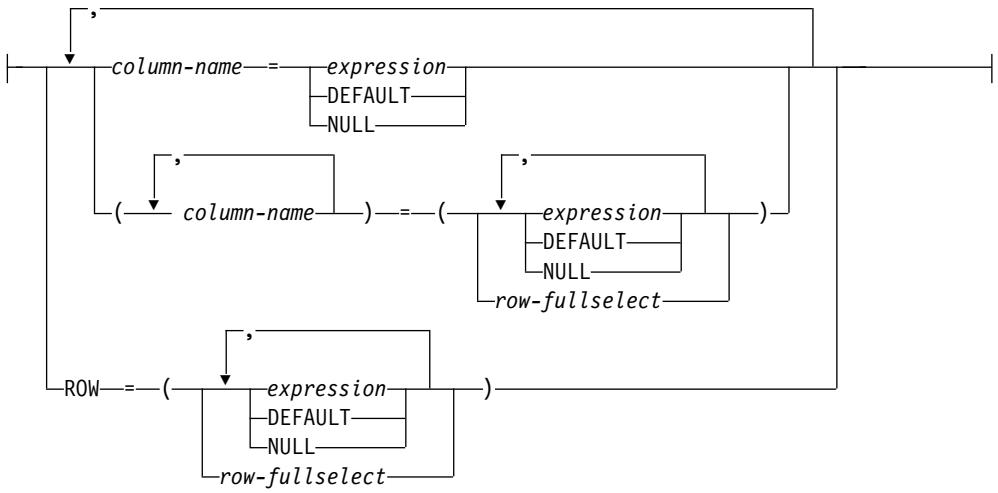
delete-operation:



insert-operation:



assignment-clause:



isolation–clause:

Description

table-name **or** *view-name*
Identifies the target of the update, insert, and delete operations of the merge. The name must identify a table or view that exists at the current server, but it must not identify a catalog table, a view of a catalog table, a read-only view, or a non-deletable view independent of any INSTEAD OF triggers defined for it.

AS *correlation-name*
Can be used within *search-condition*, *matching-condition*, or on the right side of an *assignment-clause* to designate the target table or view. The *correlation-name* is used to qualify references to the columns of the table or view. When a *correlation-name* is specified, *column-name*s can also be specified to give names

to the columns of the *table-name* or *view-name*. If a column list is specified, there must be a name in the column list for each column in the table or view. For more information, see "Correlation names" on page 140

USING *table-reference*
Specifies a set of rows as a result table to be merged into the target. If the result table is empty, a warning is returned.

ON *search-condition*
Specifies the predicates used to determine whether a row from *source-table* matches rows in the target table.

Each *column-name* in the *search-condition*, other than in a subquery, must name a column of the target table or view or the *table-reference*. When the search condition includes a subquery in which the same table is the base object of both the merge and the subquery, the subquery is completely evaluated before any rows are updated or inserted.

The *search-condition* is applied to each row of the target table and result table of the *table-reference*. For those rows of the result table of the *table-reference* where the result of the *search-condition* is true, the specified update or delete operation is performed. For those rows of the result table where the result of the *search-condition* is not true, the specified insert operation is performed.

The search-condition cannot contain a quantified subquery, IN predicate with a subselect, or EXISTS subquery. It can contain basic predicate subqueries or scalar-fullselects. It cannot contain expressions that use aggregate functions or non-deterministic scalar functions.

WHEN *matching-condition*
Specifies the condition under which the *update-operation*, *delete-operation*, *insert-operation*, or the *signal-statement* is executed. Each *matching-condition* is evaluated in order of specification. Rows for which the *matching-condition* evaluates to true are not considered in subsequent matching conditions.

MATCHED
Indicates the operation to be performed on the rows where the ON *search-condition* is true. Only UPDATE, DELETE, or *signal-statement* can be specified after THEN.

AND *search-condition*
Specifies a further search condition to be applied against the rows that matched the ON search condition for the operation to be performed after THEN.

The *search-condition* must not include a subquery in an EXISTS or IN predicate.

NOT MATCHED
Indicates the operation to be performed on the rows where the ON *search-condition* is false or unknown. Only INSERT or *signal-statement* can be specified after THEN.

AND *search-condition*
Specifies a further search condition to be applied against the rows that did not match the ON search condition for the operation to be performed after THEN.

The *search-condition* must not include a subquery in an EXISTS or IN predicate.

THEN
Specifies the operation to execute when the *matching-condition* evaluates to true.

update-operation
Specifies the update operation to be executed for the rows where the *matching-condition* evaluates to true.

assignment-clause
Specifies a list of column updates.

column-name
Identifies a column to be updated. The *column-name* must identify a column of the target table or view. The *column-name* must not identify a view column derived from a scalar function, constant, or expression. A column name must not be specified more than once.

A view column derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same MERGE statement.

ROW
Identifies all the columns of the target table or view except for columns defined with the hidden attribute. If a view is specified, none of the columns of the view may be derived from a scalar function, constant, or expression.

The number of expressions, NULLs, and DEFAULTs (or the number of result columns from a *row-fullselect*) must match the number of columns in the row.

ROW may not be specified for a view that contains a view column derived from the same column as another column of the view, because both columns cannot be updated in the same UPDATE.

expression
Indicates the new value of the column. The expression must not include an aggregate function except when it occurs within a scalar fullselect.

The expression can contain references to columns of the target *table-name* or *view-name*. For each row that is updated, the value of a target column reference in an expression is the value of the column in the row before the row is updated.

If *expression* is a reference to a single column of the source table, the source table column value may have been specified with an extended indicator variable value. The effects of such indicator variables apply to the corresponding target columns of the *assignment-clause*.

When extended indicator variables are enabled, the extended indicator variable values of DEFAULT (-5) or UNASSIGNED (-7) must not be used if expression is more complex than the following references:

- A single column of the source table
- A single host variable

DEFAULT
Specifies that the default value is assigned to the column. DEFAULT can be specified only for columns that have a default

value. For more information about default values, see the description of the DEFAULT clause in "CREATE TABLE" on page 982.

DEFAULT must be specified for a column that was defined as GENERATED ALWAYS unless OVERRIDING USER VALUE is specified to indicate that any user-specified value will be ignored and a unique system-generated value will be used. A valid value can be specified for a column that was defined as GENERATED BY DEFAULT.

NULL
Specifies the null value as the new value of the column. Specify NULL only for nullable columns.

delete-operation
Specifies the delete operation to be executed for the rows where the *matching-condition* evaluates to true.

insert-operation
Specifies the insert operation to be executed for the rows where the *matching-condition* evaluates to true.

INSERT
Introduces a list of column names and row value expressions to be used for the insert operation.

The number of values in the row value expression must equal the number of names in the implicit or explicit insert column list. The first value is inserted in the first column in the list, the second value in the second column, and so on.

(*column-name***,...)**
Specifies the columns for which insert values are provided. Each name must be a name that identifies a column of the table or view. The same column must not be identified more than once. If extended indicator variables are not enabled, a view column that is not updatable must not be identified. If extended indicator variables are not enabled and the object of the insert operation is a view with such columns, a list of column names must be specified and the list must not identify those columns. For an explanation of updatable columns in views, see "CREATE VIEW" on page 1069.

Omission of the column list is an implicit specification of a list in which every column of the table or view is identified in left-to-right order. Any columns defined with the hidden attribute are omitted. This list is established when the statement is prepared and, therefore, does not include columns that were added to a table after the statement was prepared.

VALUES
Specifies a new row to be inserted.

Each variable in the clause must identify a variable that is declared in accordance with the rules for declaring variables. A host structure cannot be used. For further information about variables, see "References to host variables" on page 149.

expression
An *expression* of the type described in "Expressions" on page 165, that does not include an aggregate function or column name.

When extended indicator variables are enabled, the extended indicator variable values of DEFAULT (-5) or UNASSIGNED (-7) must not be used if expression is more complex than the following references:

- A single column of the source table
- A single host variable
- A host variable being explicitly cast

DEFAULT
Specifies that the default value is assigned to the column. DEFAULT can be specified only for columns that have a default value. For more information about default values, see the description of the DEFAULT clause in "CREATE TABLE" on page 982.

DEFAULT must be specified for a column that was defined as GENERATED ALWAYS unless OVERRIDING USER VALUE is specified to indicate that any user-specified value will be ignored and a unique system-generated value will be used. A valid value can be specified for a column that was defined as GENERATED BY DEFAULT.

NULL
Specifies the value for a column is the null value. NULL should only be specified for nullable columns.

OVERRIDING SYSTEM VALUE or OVERRIDING USER VALUE
Specifies whether system generated values or user-specified values for a ROWID, identity, or row change timestamp column are used. If OVERRIDING SYSTEM VALUE is specified, the implicit or explicit list of columns for the INSERT or the SET clause of the UPDATE must contain a column defined as GENERATED ALWAYS. If OVERRIDING USER VALUE is specified, the implicit or explicit list of columns for the INSERT or the SET clause of the UPDATE must contain a column defined as either GENERATED ALWAYS or GENERATED BY DEFAULT.

OVERRIDING SYSTEM VALUE
Specifies that the value specified in the VALUES or SET clause for a column that is defined as GENERATED ALWAYS is used. A system-generated value is not used.

OVERRIDING USER VALUE
Specifies that the value specified in the VALUES or SET clause for a column that is defined as either GENERATED ALWAYS or GENERATED BY DEFAULT is ignored. Instead, a system-generated value is used, overriding the user-specified value.

If neither OVERRIDING SYSTEM VALUE nor OVERRIDING USER VALUE is specified:

- A value cannot be specified for a ROWID, identity, or row change timestamp column that is defined as GENERATED ALWAYS.
- A value can be specified for a ROWID, identity, or row change timestamp column that is defined as GENERATED BY DEFAULT. If a value is specified that value is assigned to the column. However, a value can be assigned to a ROWID column defined BY DEFAULT only if the specified value is a valid row ID value that was previously generated by DB2 for z/OS or DB2 for i. When a value is inserted or updated for an identity or row change timestamp column defined BY DEFAULT, the

database manager does not verify that the specified value is a unique value for the column unless the identity or row change timestamp column is the sole key in a unique constraint or unique index. Without a unique constraint or unique index, the database manager can guarantee unique values only among the set of system-generated values as long as NO CYCLE is in effect.

If a value is not specified the database manager generates a new value.

signal-statement
Specifies the SIGNAL statement that is to be executed to return an error when the *matching-condition* evaluates to true. See "SIGNAL" on page 1407.

ELSE IGNORE
Specifies that no action is to be taken when the *matching-condition* for all WHEN clauses is false for a row in the USING *table-reference*. No action is taken in this case whether or not ELSE IGNORE is specified.

ATOMIC or NOT ATOMIC
Specifies how to handle errors

ATOMIC
Specifies that if an error occurs during a *update-operation*, *delete-operation*, or *insert-operation*, the entire MERGE statement is rolled back.

NOT ATOMIC
Specifies that if an error occurs during a *update-operation*, *delete-operation*, or *insert-operation*, only that *update-operation*, *delete-operation*, or *insert-operation* is rolled back.

STOP ON SQL EXCEPTION
Specifies that if an error occurs during a *update-operation*, *delete-operation*, or *insert-operation*, the processing of the MERGE statement stops.

CONTINUE ON SQL EXCEPTION
Specifies that if an error occurs during a *update-operation*, *delete-operation*, or *insert-operation*, the processing of the MERGE statement continues.

isolation-clause
Specifies the isolation level to be used for this statement.

WITH

Introduces the isolation level, which may be one of:
- *RR* Repeatable read
- *RS* Read stability
- *CS* Cursor stability
- *UR* Uncommitted read
- *NC* No commit

If *isolation-clause* is not specified the default isolation is used. See "isolation-clause" on page 693 for a description of how the default is determined.

concurrent-access-resolution-clause
Specifies the concurrent access resolution to use for the select statement. For more information, see "concurrent-access-resolution-clause" on page 695.

MERGE Rules

- More than one *update-operation*, *delete-operation*, *insert-operation*, or *signal-statement* can be specified in a single MERGE statement.
- Each row in the target can only be operated on once. A row in the target can only be identified as MATCHED with one row in the result table of the *table-reference*. A nested SQL operation (RI or trigger except INSTEAD OF trigger) cannot specify the target table (or a table within the same hierarchy) as a target of an UPDATE, DELETE, INSERT, or MERGE statement.

For other rules that affect the update, insert, or delete portion of the MERGE statement, see the "Rules" section of the corresponding statement description.

Extended indicator variable usage: If enabled, indicator variable values other than positive values and 0 (zero) through -7 must not be set. The DEFAULT and UNASSIGNED extended indicator variable values must not appear in contexts where they are not supported.

Extended indicator variables: In an insert portion of the MERGE statement, the extended indicator value of UNASSIGNED has the effect of setting the column to its default value.

MERGE restriction: If the target table of the MERGE statement has triggers or is the parent in a referential integrity constraint, the *update-operation* or *insert-operation* must not contain a global variable, a function, or a subselect.

Notes

Logical order of processing: For a NOT ATOMIC MERGE statement, each source row is processed independently as if a separate MERGE statement were executed for each source row. For example, a source row that causes an update of a target row, will fire any triggers (including statement level triggers) when the update of the row is performed. Thus, if 5 rows are updated, any update triggers (including statement level update triggers) will be fired 5 times.

For an ATOMIC MERGE statement, the source rows are processed as if a set of rows is processed by each WHEN clause. Thus, if 5 rows are updated, any row level update triggers will be fired 5 times, but *n* statement level update triggers will be fired, where *n* is the number of WHEN clauses that contain an UPDATE, including any WHEN clauses that contain an UPDATE that did not process any of the source rows. For ATOMIC MERGE, the logical order of processing is:

1. Determine the set of rows to be processed from the source and target. If any of the special registers CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP are used in this statement, only one clock reading is done for the whole statement.
2. Use the ON clause to classify these rows as either MATCHED or NOT MATCHED.
3. Evaluate any *matching-condition* in the WHEN clauses.
4. Evaluate any expression in any *assignment-clause* and *insert-operation*.
5. Execute each *signal-statement*.
6. Apply each *update-operation*, *delete-operation*, or *insert-operation* to the applicable rows in the order of specification. The triggers activated by each *update-operation*, *delete-operation*, or *insert-operation* are executed. Statement level triggers are activated even if no rows satisfy the *update-operation*, *delete-operation*,

or *insert-operation*. Each *update-operation*, *delete-operation*, or *insert-operation* can affect the triggers of each subsequent *update-operation*, *delete-operation*, or *insert-operation*.

Number of rows updated: After executing a MERGE statement, the ROW_COUNT statement information item in the SQL Diagnostics Area (or SQLERRD(3) of the SQLCA) is the number of rows operated on by the MERGE statement, excluding rows identified by the ELSE IGNORE clause. The ROW_COUNT item and SQLERRD(3) does not include the number of rows that were operated on as a result of triggers. The value in the DB2_ROW_COUNT_SECONDARY statement information item (or SQLERRD(5) of the SQLCA) includes the number of these rows.

For a description of ROW_COUNT and DB2_ROW_COUNT_SECONDARY, see "GET DIAGNOSTICS" on page 1194. For a description of the SQLCA, see Appendix C, "SQLCA (SQL communication area)," on page 1513.

GET DIAGNOSTICS considerations: If a MERGE statement completes with one or more errors, the GET DIAGNOSTICS statement can be used after the MERGE statement to check which input row(s) failed. The GET DIAGNOSTICS *statement-information-item*, NUMBER, indicates the number of conditions (errors of warnings) detected by execution of the MERGE statement. For each condition, the GET DIAGNOSTICS *condition-information-item*, DB2_ROW_NUMBER, indicates the input source row that caused an error.

Inserted row cannot also be updated: No attempt is made to update a row in the target that did not already exist before the MERGE statement was executed; that is, there are no updates of rows that were inserted by the MERGE statement.

Concurrent row changes in MERGE target: MERGE processing determines affected rows in the MERGE target before performing any *update-operations*, *delete-operations*, or *insert-operations*. Unless using a restrictive isolation level such as repeatable read, concurrent processes could insert or modify rows in the MERGE target between the time when the set of affected target rows is determined and when a specific row *update-operation*, *delete-operation*, or *insert-operation* is processed. Such concurrent activity could produce an error. For example, MERGE processing could determine that a source row does not exist in the target where a column value in the target has a unique key constraint. Before the MERGE attempts to insert a new row based on the source data, a concurrent process could insert a row with the same key value. This would cause a duplicate key error when the MERGE processing attempts to insert its row.

Locking: If COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) is specified, one or more exclusive locks are acquired during the execution of a successful MERGE statement. Until the locks are released by a commit or rollback operation, an inserted or updated row can only be accessed by:

- The application process that performed the insert or update
- Another application process using COMMIT(*NONE) or COMMIT(*CHG) through a read-only operation

The locks can prevent other application processes from performing operations on the table. For further information about locking, see the description of the "COMMIT" on page 824, "ROLLBACK" on page 1338, and "LOCK TABLE" on page 1272 statements. Also, see "Isolation level" on page 26 and Database Programming.

A maximum of 500 000 000 rows can be acquired in any single MERGE statement when COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) was specified. The number of row locks includes any rows inserted, updated, or deleted in the MERGE target and any rows inserted, updated, or deleted under the same commitment definition as a result of a trigger. The number of row locks also includes source rows referenced by the USING *table-reference* is COMMIT(*ALL) is specified.

NOT ATOMIC processing: When NOT ATOMIC is specified, the rows of source data are processed separately. Any reference to special registers (such as CURRENT TIMESTAMP) in the MERGE statement are evaluated as each row or source data is processed. Statement level triggers are activated as each row of source data is processed.

If an error occurs during the operation for a row of source data, the row being processed at the time of the error is not inserted, updated, or deleted. Processing of an individual row is an atomic operation. Any other changes previously made during the processing of the MERGE statement are not rolled back. If CONTINUE ON EXCEPTION is specified, execution continues with the next row to be processed.

Examples

Example 1: For activities whose description has changed, update the description in the archive table. For new activities, insert into the archive table. The archive and activities tables both have activity as a primary key.

```
MERGE INTO archive ar
  USING (SELECT activity, description FROM activities) ac
  ON (ar.activity = ac.activity)
  WHEN MATCHED THEN
    UPDATE SET description = ac.description
  WHEN NOT MATCHED THEN
    INSERT (activity, description) VALUES(ac.activity, ac.description)
```

Example 2: Using the shipment table, merge rows into the inventory table, increasing the quantity by part count in the shipment table for rows that match; else insert the new *partno* into the inventory table.

```
MERGE INTO inventory AS in
  USING (SELECT partno, description, count FROM shipment
           WHERE shipment.partno IS NOT NULL) AS sh
  ON (in.partno = sh.partno)
  WHEN MATCHED THEN
    UPDATE SET description = sh.description,
               quantity = in.quantity + sh.count
  WHEN NOT MATCHED THEN
    INSERT (partno, description, quantity)
           VALUES (sh.partno, sh.description, sh.count)
```

Example 3: Using the transaction table, merge rows into the account table, updating the balance from the set of transactions against an account ID and inserting new accounts from the consolidated transactions where they do not already exist.

```
MERGE INTO account AS a
    USING (SELECT id, SUM(amount) sum_amount FROM transaction
             GROUP BY id) AS t
    ON a.id = t.id
    WHEN MATCHED THEN
      UPDATE SET balance = a.balance + t.sum_amount
    WHEN NOT MATCHED THEN
      INSERT (id, balance) VALUES (t.id, t.sum_amount)
```

Example 4: Using the transaction_log table, merge rows into the employee_file table, updating the phone and office columns with the latest transaction_log row based on the transaction time, and inserting a new employee_file row where the row does not already exist.

```
MERGE INTO employee_file AS e
  USING (SELECT empid, phone, office
              FROM (SELECT empid, phone, office,
                           ROW_NUMBER() OVER (PARTITION BY empid
                                              ORDER BY transaction_time DESC) rn
                    FROM transaction_log) AS nt
              WHERE rn = 1) AS t
  ON e.empid = t.empid
  WHEN MATCHED THEN
    UPDATE SET (phone, office) = (t.phone, t.office)
  WHEN NOT MATCHED THEN
    INSERT (empid, phone, office)
           VALUES(t.empid, t.phone, t.office)
```

Example 5: Using dynamically supplied values for an employee row, update the master employee table if the data corresponds to an existing employee, or insert the row if the data is for a new employee. The following example is a fragment of code from a C program.

```
hv1 =
"MERGE INTO employee AS t
   USING (VALUES(CAST(? AS CHAR(6)), CAST(? AS VARCHAR(12)),
                      CAST(? AS CHAR(1)), CAST(? AS VARCHAR(15)),
                      CAST(? AS SMALLINT), CAST(? AS INTEGER)))
          s (empno, firstnme, midinit, lastname, edlevel, salary)
   ON t.empno = s.empno
   WHEN MATCHED THEN
     UPDATE SET salary = s.salary
   WHEN NOT MATCHED THEN
     INSERT (empno, firstnme, midinit, lastname, edlevel, salary)
            VALUES (s.empno, s.firstnme, s.midinit, s.lastname, s.edlevel,
                    s.salary)";
EXEC SQL PREPARE s1 FROM :hv1;
EXEC SQL EXECUTE s1 USING :hv2, :hv3, :hv4, :hv5, :hv6, :hv7;
```

Example 6: Update the list of activities organized by Group A in the archive table. Delete all outdated activities and update the activities information (description and date) in the archive table if they have been changed. For new upcoming activities, insert into the archive. Signal an error if the data of the activity is not known. The date of the activities in the archive table must be specified. Each group has an activities table. For example, activities_groupA contains all activities that they organize, and the archive table contains all upcoming activities organized by different groups in a company. The archive table has (group, activity) as the primary key, and data is not nullable. All activities tables have activity as the primary key. The last_modified column in the archive is defined with CURRENT TIMESTAMP as the default value.

```
MERGE INTO archive ar
  USING (SELECT activity, description, date, last_modified
           FROM activities_groupA) ac
  ON (ar.activity = ac.activity) AND ar.group = 'A'
  WHEN MATCHED AND ac.date IS NULL THEN
    SIGNAL SQLSTATE '70001'
           SET MESSAGE_TEXT = 'Activity cannot be modified.  Reason: date is not known'
  WHEN MATCHED and ac.date < CURRENT DATE THEN
    DELETE
  WHEN MATCHED AND as.last_modified < ac.last_modified THEN
    UPDATE SET (description, date, last_modified)
               = (ac.description, ac.date, DEFAULT)
  WHEN NOT MATCHED AND ac.date IS NULL THEN
    SIGNAL SQLSTATE '70002'
           SET MESSAGE_TEXT = 'Activity cannot be inserted.  Reason: date is not known'
```

```
                        WHEN NOT MATCHED AND ac.date >= CURRENT DATE THEN
                          INSERT (group, activity, description, date)
                                 VALUES ('A', ac.activity, ac.description, ac.date)
                        ELSE IGNORE
```

OPEN

The OPEN statement opens a cursor so that it can be used to fetch rows from its result table.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

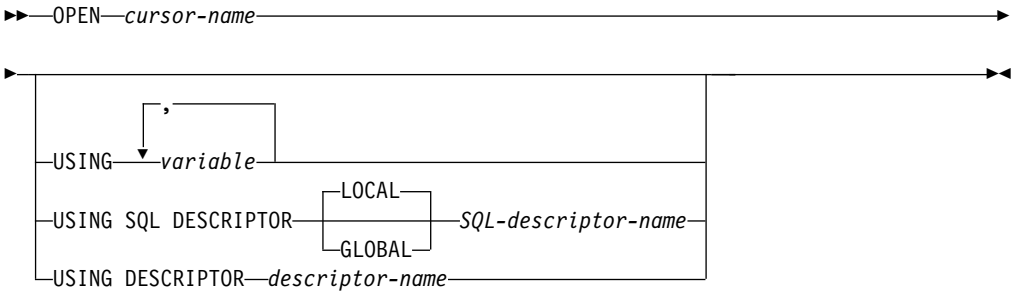
Authorization

If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

See "DECLARE CURSOR" on page 1079 for the authorization required to use a cursor.

Syntax



Description

cursor-name

Identifies the cursor to be opened. The *cursor-name* must identify a declared cursor as explained in the Notes for the DECLARE CURSOR statement. When the OPEN statement is executed, the cursor must be in the closed state.

The SELECT statement associated with the cursor is either:

- The *select-statement* specified in the DECLARE CURSOR statement, or
- The prepared *select-statement* identified by the *statement-name* specified in the DECLARE CURSOR statement. If the statement has not been successfully prepared, or is not a *select-statement*, the cursor cannot be successfully opened.

The result table of the cursor is derived by evaluating the SELECT statement. The evaluation uses the current values of any special registers specified in the SELECT statement and the current values of any variables specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table can be derived during the execution of the OPEN statement and a temporary table can be created to hold them; or they can be derived

during the execution of subsequent FETCH statements. In either case, the cursor is placed in the open state and positioned before the first row of its result table. If the table is empty the position of the cursor is effectively "after the last row."

USING
Introduces a list of variables whose values are substituted for the parameter markers (question marks) of a prepared statement. For an explanation of parameter markers, see "PREPARE" on page 1293.

- If a *statement-name* is specified in the DECLARE CURSOR statement that includes parameter markers, USING must be used. If the prepared statement does not include parameter markers, USING is ignored.
- If a *select-statement* is specified in the DECLARE CURSOR statement, USING may be used to override the variable values. For more information, see Variable value override.

If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, you must use USING. If the prepared statement does not include parameter markers, USING is ignored.

variable,...
Identifies host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and variables. A reference to a host structure is replaced by a reference to each of its variables. The resulting number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement.

A global variable may only be used if the current connection is a local connection (not a DRDA connection).

USING SQL DESCRIPTOR *SQL-descriptor-name*
Identifies an SQL descriptor.

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "SET DESCRIPTOR" on page 1361 for an explanation of the information in the SQL descriptor.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of input variables.

Before the OPEN statement is processed, the user must set the following fields in the SQLDA. (The rules for REXX are different. For more information see Embedded SQL Programming.)

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA

- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables

The SQLDA must have enough storage to contain all SQLVAR occurrences. If LOBs or distinct types are present in the results, there must be additional SQLVAR entries for each parameter. For more information about the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see Appendix D, "SQLDA (SQL descriptor area)," on page 1523.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement. The *n*th variable described by the SQLDA corresponds to the *n*th parameter marker in the prepared statement.

Note that because RPG/400 does not provide the facility for setting pointers and the SQLDA uses pointers to locate the appropriate variables, you will have to set these pointers outside your RPG/400 application.

Notes

Closed state of cursors: All cursors in a program are in the closed state when:

- The program is called:
  - If CLOSQLCSR(*ENDPGM) is specified, all cursors are in the closed state each time the program is called.
  - If CLOSQLCSR(*ENDSQL) is specified, all cursors are in the closed state only the first time the program is called as long as one SQL program remains on the call stack.
  - If CLOSQLCSR(*ENDJOB) is specified, all cursors are in the closed state only the first time the program is called as long as the job remains active.
  - If CLOSQLCSR(*ENDMOD) is specified, all cursors are in the closed state each time the module is initiated.
  - If CLOSQLCSR(*ENDACTGRP) is specified, all cursors are in the closed state only the first time the module in the program is initiated in the activation group.
- A program starts a new unit of work by executing a COMMIT statement without a HOLD option. Cursors declared with the HOLD option are not closed by a COMMIT statement without a HOLD option. A COMMIT HOLD statement does not close cursors whether they are declared with a HOLD option or not.
- A program starts a new unit of work by executing a ROLLBACK statement without a HOLD option. A ROLLBACK HOLD statement does not close cursors whether they are declared with a HOLD option or not.
- A CONNECT (Type 1) statement was executed.

A cursor can also be in the closed state because:

- A CLOSE statement was executed.
- A DISCONNECT statement disconnected the connection with which the cursor was associated.
- The connection with which the cursor was associated was in the release-pending state and a successful COMMIT occurred.
- A CONNECT (Type 1) statement was executed.

To retrieve rows from the result table of a cursor, the FETCH statement must be executed when the cursor is open. The only way to change the state of a cursor from closed to open is to execute an OPEN statement.

Effect of temporary tables: If the result table of a cursor is not read-only, its rows are derived during the execution of subsequent FETCH statements. The same method may be used for a read-only result table. However, if a result table is read-only, DB2 for i may choose to use the temporary table method instead. With this method the entire result table is inserted into a temporary table during the execution of the OPEN statement. When a temporary table is used, the results of a program can differ in several ways:

- An error can occur during OPEN that would otherwise not occur until some later FETCH statement.
- The INSERT, UPDATE, and DELETE statements that are executed while the cursor is open cannot affect the result table.
- Any NEXT VALUE expressions in the SELECT statement are evaluated for every row of the result table during OPEN. Thus, sequence values are generated, for every row of the result table during OPEN.
- Any functions are evaluated for every row of the result table during OPEN. Thus, any external actions and SQL statements that modify SQL data within the functions are performed for every row of the result table during OPEN.

Conversely, if a temporary table is not used, INSERT, UPDATE, and DELETE statements executed while the cursor is open can affect the result table, and any NEXT VALUE expressions and functions in the SELECT statement are evaluated as each row is fetched. The effect of such operations is not always predictable. For example, if cursor CUR is positioned on a row of its result table defined as SELECT * FROM T, and a row is inserted into T, the effect of that insert on the result table is not predictable because its rows are not ordered. A subsequent FETCH CUR might or might not retrieve the new row of T.

Parameter marker replacement: When the SELECT statement of the cursor is evaluated, each parameter marker in the statement is effectively replaced by its corresponding variable. The replacement of a parameter marker is an assignment operation in which the source is the value of the variable, and the target is a variable within the database manager. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker. For the rules that affect parameter markers, see Table 105 on page 1300.

Let V denote a variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:

- V must be compatible with the target.
- If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, the value of V must not be null.

However, unlike the rules for assigning a value to a column:

- If V is a string, the value will be truncated (without an error), if its length is greater than the length attribute of the target.

When the SELECT statement of the cursor is evaluated, the value used in place of P is the value of the target variable for P. For example, if V is CHAR(6), and the target is CHAR(8), the value used in place of P is the value of V padded with two blanks.

The USING clause is intended for a prepared SELECT statement that contains parameter markers. However, it can also be used when the SELECT statement of the cursor is part of the DECLARE CURSOR statement. In this case the OPEN statement is executed as if each variable in the SELECT statement were a parameter marker, except that the attributes of the target variables are the same as the attributes of the variables in the SELECT statement. The effect is to override the values of the variables in the SELECT statement of the cursor with the values of the variables specified in the USING clause.

Variable value override: The USING clause is intended for a prepared SELECT statement that contains parameter markers. However, it can also be used when the SELECT statement of the cursor is part of the DECLARE CURSOR statement. In this case, the OPEN statement is executed as if each variable in the SELECT statement were a parameter marker, except that the attributes of the target variables are the same as the attributes of the variables in the SELECT statement. The effect is to override the values of the variables in the SELECT statement of the cursor with the values of the variables specified in the USING clause.

Examples

Example 1: Write the embedded statements in a COBOL program that will:

1. Define a cursor C1 that is to be used to retrieve all rows from the DEPARTMENT table for departments that are administered by (ADMRDEPT) department ‘A00'
2. Place the cursor C1 before the first row to be fetched.

```
EXEC SQL  DECLARE C1 CURSOR FOR
           SELECT DEPTNO, DEPTNAME, MGRNO FROM DEPARTMENT
              WHERE ADMRDEPT = 'A00'  END-EXEC.

EXEC SQL  OPEN C1  END-EXEC.
```

Example 2: Code an OPEN statement to associate a cursor DYN_CURSOR with a dynamically defined *select-statement* in a C program. Assume each prepared *select-statement* always defines two items in its select list with the first item having a data type of integer and the second item having a data type of VARCHAR(64). (The related host variable definitions, PREPARE statement, and DECLARE CURSOR statement are also shown in the example below.)

```
EXEC SQL  BEGIN DECLARE SECTION;
   static short hv_int;
   char hv_vchar64[64];
   char stmt1_str[200];
EXEC SQL  END DECLARE SECTION;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;

EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING :hv_int, :hv_vchar64;
```

Example 3: Code an OPEN statement as in example 3, but in this case the number and data types of the items in the select statement are not known.

```
EXEC SQL  BEGIN DECLARE SECTION;
   char stmt1_str[200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING DESCRIPTOR :sqlda;
```

PREPARE

The PREPARE statement creates an executable form of an SQL statement from a character-string form of the statement. The character-string form is called a *statement string*, and the executable form is called a *prepared statement*.

Invocation

This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The authorization rules are the same as those defined for the SQL statement specified by the PREPARE statement. For example, see "select-statement" on page 682 for the authorization rules that apply when a SELECT statement is prepared.

If DLYPRP(*NO) is specified on the CRTSQLxxx command, the authorization checking is performed when the statement is prepared, except:

- If a DROP SCHEMA statement is prepared, privileges on all objects in the schema are not checked until the statement is executed.
- If a DROP TABLE statement is prepared, privileges on all views, indexes, and logical files that reference the table are not checked until the statement is executed.
- If a DROP VIEW statement is prepared, privileges on all views that reference the view are not checked until the statement is executed.
- If a CREATE TRIGGER statement is prepared, privileges on objects referenced in the *triggered-action* are not checked until the statement is executed.
- If a DROP, COMMENT, or LABEL of a FUNCTION, PROCEDURE, SEQUENCE, TYPE, TRIGGER, VARIABLE, or XSROBJECT statement is prepared, authorities are not checked until the statement is executed.
- If a GRANT or REVOKE statement is prepared, authorities are not checked until the statement is executed.

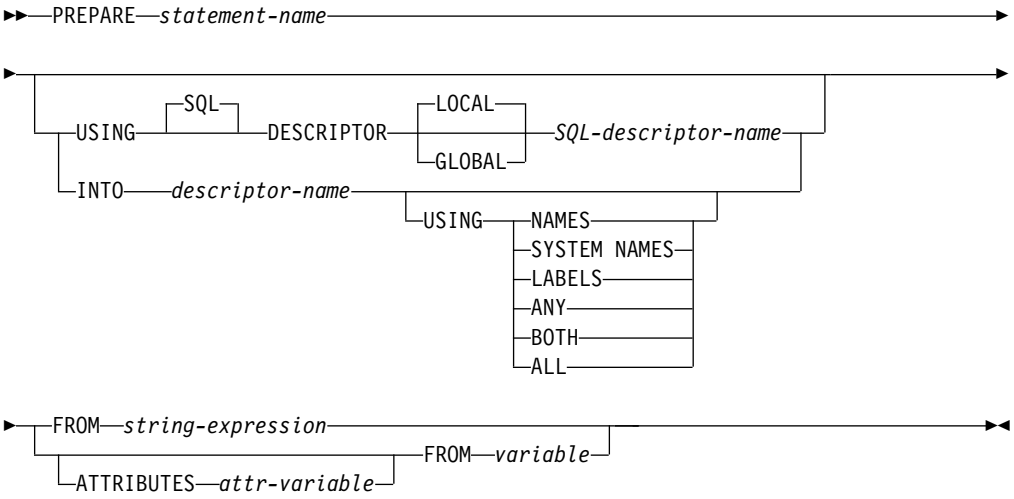
If DLYPRP(*YES) is specified on the CRTSQLxxx command, all authorization checking is deferred until the statement is executed or used in an OPEN statement.

The authorization ID of the statement is the run-time authorization ID unless DYNUSRPRF(*OWNER) was specified on the CRTSQLxxx command when the program was created. For more information, see "Authorization IDs and authorization names" on page 68.

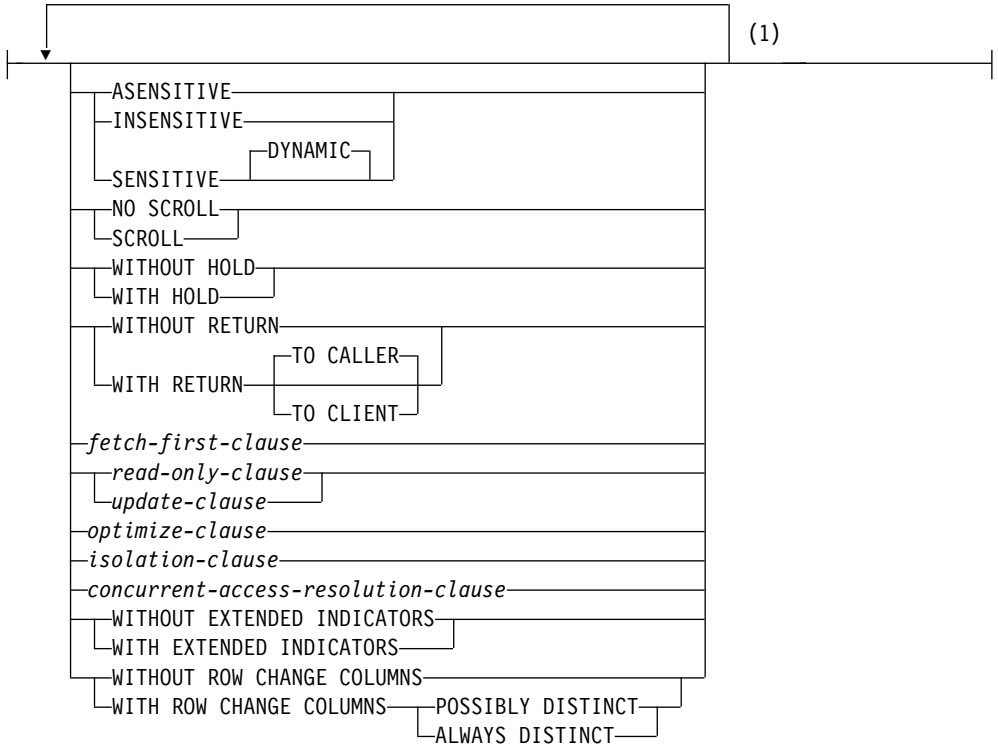
If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



attribute-string:



Notes:

1 The same clause must not be specified more than once. If the options are not specified, their defaults are whatever was specified for the corresponding options in an associated DECLARE CURSOR and the prepared SELECT statement.

Description

statement-name
Names the prepared statement. If the name identifies an existing prepared statement, that prepared statement is destroyed if:

- it was prepared in the same instance of the same program, or
- CLOSQLCSR(*ENDJOB), CLOSQLCSR(*ENDACTGRP), or CLOSQLCSR(*ENDSQL) are specified on the CRTSQLxxx commands associated with both prepared statements.

The name must not identify a prepared statement that is the SELECT statement of an open cursor of this instance of the program.

USING SQL DESCRIPTOR *SQL-descriptor-name*
Identifies an SQL descriptor. If USING is specified, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQL descriptor specified by the *SQL-descriptor-name*. Thus, the PREPARE statement:

```
EXEC SQL PREPARE S1 USING SQL DESCRIPTOR :sqldescriptor FROM :V1;
```

is equivalent to:

```
EXEC SQL PREPARE S1 FROM :V1;
EXEC SQL DESCRIBE S1 USING SQL DESCRIPTOR :sqldescriptor;
```

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

See "GET DESCRIPTOR" on page 1182 for an explanation of the information that is placed in the SQL descriptor.

INTO
If INTO is used, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQLDA specified by the *descriptor-name*. Thus, the PREPARE statement:

```
EXEC SQL PREPARE S1 INTO :SQLDA FROM :V1;
```

is equivalent to:

```
EXEC SQL PREPARE S1 FROM :V1;
EXEC SQL DESCRIBE S1 INTO :SQLDA;
```

descriptor-name
Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, "SQLDA (SQL descriptor area)," on page 1523. Before the PREPARE statement is executed, the following variable in the SQLDA must be set (The rules for REXX are different. For more information, see the Embedded SQL Programming topic collection.) :

SQLN
Indicates the number of variables represented by SQLVAR. (SQLN provides the dimension of the SQLVAR array.) SQLN must be set to a value greater than or equal to zero before the PREPARE statement is

executed. For information about techniques to determine the number of occurrences required, see "Determining how many SQLVAR occurrences are needed" on page 1526.

See "DESCRIBE" on page 1127 for an explanation of the information that is placed in the SQLDA.

USING
Specifies what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist or a name is longer than 30, SQLNAME is set to length 0.

NAMES
Assigns the name of the column. This is the default. For a prepared statement where the names are explicitly specified in the select-list, the name specified is returned.

SYSTEM NAMES
Assigns the system column name of the column.

LABELS
Assigns the label of the column. (Column labels are defined by the LABEL statement.) Only the first 20 bytes of the label are returned.

ANY
Assigns the column label. If the column has no label, the label is the column name.

BOTH
Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 2*n* or 3*n*(where *n* is the number of columns in the table or view). The first *n* occurrences of SQLVAR contain the column names. Either the second or third *n* occurrences contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

If the same SQLDA is used on a subsequent FETCH statement, set SQLN to *n* after the PREPARE is complete.

ALL
Assigns the label, column name, and system column name. In this case three or four occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 3*n* or 4*n* (where *n* is the number of columns in the result table). The first *n* occurrences of SQLVAR contain the system column names. The second or third *n* occurrences contain the column labels. The third or fourth *n* occurrences contain the column names if they are different from the system column name. If there are no distinct types, the labels are returned in the second set of SQLVAR entries and the column names are returned in the third set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries and the column names are returned in the fourth set of SQLVAR entries.

If the same SQLDA is used on a subsequent FETCH statement, set SQLN to *n* after the PREPARE is complete.

ATTRIBUTES *attr-variable*

Specifies the attributes for this cursor that are in effect if a corresponding attribute has not been specified as part of the outermost fullselect of the associated SELECT statement. If attributes are specified for the outermost fullselect, they are used instead of the corresponding attributes specified on the PREPARE statement. In turn, if attributes are specified in the PREPARE statement, they are used instead of the corresponding attributes specified on a DECLARE CURSOR statement.

All attributes other than USE CURRENTLY COMMITTED and WAIT FOR OUTCOME are ignored if the prepared statement is not a *select-statement*.

attr-variable must identify a character-string or Unicode graphic variable that is declared in the program in accordance with the rules for declaring string variables. *attr-variable* must be a string variable (either fixed-length or varying-length) that has a length attribute that does not exceed the maximum length of a VARCHAR. Leading and trailing blanks are removed from the value of the variable. The variable must contain a valid *attribute-string*.

An indicator variable can be used to indicate whether attributes are actually provided on the PREPARE statement. Thus, applications can use the same PREPARE statement regardless of whether attributes need to be specified or not. The options that can be specified as part of the *attribute-string* are as follows:

ASENSITIVE, SENSITIVE, or INSENSITIVE
Specifies whether the cursor is asensitive, sensitive, or insensitive to changes. For more information, see "DECLARE CURSOR" on page 1079.

If SENSITIVE is specified, then a *fetch-first-clause* must not be specified. If INSENSITIVE is specified, then an *update-clause* must not be specified.

NO SCROLL or SCROLL
Specifies whether the cursor is scrollable or not scrollable. For more information, see "DECLARE CURSOR" on page 1079.

WITHOUT HOLD or WITH HOLD
Specifies whether the cursor should be prevented from being closed as a consequence of a commit operation. For more information, see "DECLARE CURSOR" on page 1079.

WITHOUT RETURN or WITH RETURN
Specifies whether the result table of the cursor is intended to be used as a result set that will be returned from a procedure. For more information, see "DECLARE CURSOR" on page 1079.

fetch-first-clause
Specifies that a maximum number of rows should be retrieved. For more information, see "fetch-first-clause" on page 672.

read-only-clause **or** *update-clause*
Specifies whether the result table is read-only or updatable. The *update-clause* clause must be specified without column names (FOR UPDATE). For more information, see "read-only-clause" on page 691 and "update-clause" on page 690.

optimize-clause
Specifies that the database manager should assume that the program does not intend to retrieve more than *integer* rows from the result table. For more information, see "optimize-clause" on page 692.

isolation-clause
Specifies an isolation level at which the select statement is executed. For more information, see "isolation-clause" on page 693.

concurrent-access-resolution-clause
Specifies the concurrent access resolution to use for the select statement. For more information, see "concurrent-access-resolution-clause" on page 695.

WITHOUT EXTENDED INDICATORS or WITH EXTENDED INDICATORS
Specifies whether the values provided for indicator variables during execution of an INSERT or UPDATE follow standard SQL semantics for indicating NULL values, or may use extended capabilities to indicate the assignment of a DEFAULT or UNASSIGNED value.

WITH EXTENDED INDICATORS must only be specified when the statement is an INSERT using VALUES form of the INSERT statement, an UPDATE statement, or when the statement contains an INSERT using VALUES form of the INSERT statement.

WITHOUT ROW CHANGE COLUMNS or WITH ROW CHANGE COLUMNS POSSIBLY DISTINCT or WITH ROW CHANGE COLUMNS ALWAYS DISTINCT
Specifies whether additional column(s) should be added to the result set of a prepared *select-statement* that can be subsequently be used to identify whether a value of a column in the row might have changed. Additional row change columns are only added if a single table (or an updatable view) is referenced in the outermost subselect. The DESCRIBE and GET DESCRIPTOR statements will indicate which rows have been added.

WITHOUT ROW CHANGE COLUMNS
Row change columns are not added to the result set. This is the default.

WITH ROW CHANGE COLUMNS POSSIBLY DISTINCT
Row change columns are added to the result set even if they do not uniquely represent a single row. The columns added can be used to determine whether a value of a column in the row might have changed since it was originally fetched.

- If the row change column values have not changed since they were first fetched, then no columns of the row have been changed since they were first fetched.
- If the row change column values have changed since they were first fetched, then columns of the row may or may not have changed since the row change values are not guaranteed to represent a single row.

WITH ROW CHANGE COLUMNS ALWAYS DISTINCT
Row change columns are added to the result set only if they uniquely represent a single row. Otherwise, no row change columns are added to the result set. The columns added can be used to determine whether a value of a column in the row has changed since it was originally fetched. (Note that a table requires a row change timestamp column to guarantee that the row change columns of a row uniquely identify a single row.)

- If the row change column values have not changed since they were first fetched, then no columns of the row have been changed since they were first fetched.

- If the row change column values have changed since they were first fetched, then columns of the row have changed.

A warning is returned (SQLSTATE 0168T) if WITH ROW CHANGE COLUMNS ALWAYS DISTINCT is specified and the database manager is unable to return distinct row change columns.

FROM

Introduces the statement string. The statement string is the value of the specified *string-expression* or the identified *variable*.

string-expression

A *string-expression* is any PL/I *string-expression* that yields a character string. SQL expressions that yield a character string are not allowed. A *string-expression* is only allowed in PL/I.

variable

Identifies a *variable* that is declared in the program in accordance with the rules for declaring character-string or Unicode graphic variables. An indicator variable must not be specified.

A global variable may only be used if the current connection is a local connection (not a DRDA connection).

The statement string must be one of the following SQL statements:

ALLOCATE CURSOR	HOLD LOCATOR	SET CURRENT DEBUG MODE
ALTER	INSERT	SET CURRENT DECFLOAT ROUNDING MODE
ASSOCIATE LOCATORS	LABEL	SET CURRENT DEGREE
CALL	LOCK TABLE	SET CURRENT IMPLICIT XMLPARSE OPTION
COMMENT	MERGE	SET ENCRYPTION PASSWORD
COMMIT	REFRESH TABLE	SET PATH
CREATE	RELEASE SAVEPOINT	SET SCHEMA
DECLARE GLOBAL TEMPORARY TABLE	RENAME	SET SESSION AUTHORIZATION
DELETE	REVOKE	SET TRANSACTION
DROP	ROLLBACK	SET variable[110]
FREE LOCATOR	SAVEPOINT	UPDATE
GRANT	*select-statement*	VALUES INTO

The statement string must not:

- Begin with EXEC SQL.
- End with END-EXEC or a semicolon.
- Include references to variables.

Notes

Parameter markers: Although a statement string cannot include references to variables, it may include *parameter markers*. These can be replaced by the values of variables when the prepared statement is executed. A parameter marker is a question mark (?) that is used where a variable could be used if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see "OPEN" on page 1287 and "EXECUTE" on page 1166.

110. The target of the SET variable statement must be a global variable.

There are two types of parameter markers:

Typed parameter marker
A parameter marker that is specified along with its target data type. It has the general form:

```
CAST(? AS data-type)
```

This notation is not a function call, but a "promise" that the type of the parameter at run time will be of the data type specified or some data type that can be converted to the specified data type. For example, in:

```
UPDATE EMPLOYEE
  SET LASTNAME = TRANSLATE(CAST(? AS VARCHAR(12)))
 WHERE EMPNO = ?
```

the value of the argument of the TRANSLATE function will be provided at run time. The data type of that value will either be VARCHAR(12), or some type that can be converted to VARCHAR(12). For more information, refer to "CAST specification" on page 185.

Untyped parameter marker
A parameter marker that is specified without its target data type. It has the form of a single question mark. The data type of an untyped parameter marker is provided by context. For example, the untyped parameter marker in the predicate of the above update statement is the same as the data type of the EMPNO column.

Typed parameter markers can be used in dynamic SQL statements wherever a variable is supported and the data type is based on the promise made in the CAST function.

Untyped parameters markers can be used in dynamic SQL statements in selected locations where variables are supported. These locations and the resulting data type are found in the following tables:

Table 105. Untyped Parameter Marker Usage in Expressions (Including Select List, CASE, and VALUES)

Untyped Parameter Marker Location	Data Type
Alone in a select list that is not in a subquery	Error
Alone in a select list that is in an EXISTS subquery	Error
Alone in a select list that is in a subquery	The data type of the other operand of the subquery.[111]
Both operands of a single arithmetic operator, after considering operator precedence and order of operation rules. Includes cases such as: `? + ? + 10`	DECFLOAT(34)
One operand of a single operator in an arithmetic expression (not a datetime expression) Includes cases such as: `? + ? * 10`	The data type of the other operand.

Table 105. Untyped Parameter Marker Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

Untyped Parameter Marker Location	Data Type
Labelled duration within a datetime expression. (Note that the portion of a labelled duration that indicates the type of units cannot be a parameter marker.)	DECIMAL(15,0)
Any other operand of a datetime expression (for instance 'timecol + ?' or '? - datecol').	Error
Both operands of a CONCAT operator	DBCLOB(1G) CCSID 1200
One operand of a CONCAT operator when the other operand is a non-CLOB character data type	VARCHAR(32740) with the same CCSID as the other operand
One operand of a CONCAT operator, when the other operand is a non-DBCLOB graphic data type	VARGRAPHIC(16370) with the same CCSID as the other operand
One operand of a CONCAT operator when the other operand is a non-BLOB binary type	VARBINARY(32740)
One operand of a CONCAT operator, when the other operand is a large object string	Same as that of the other operand
The expression following the CASE keyword in a simple CASE expression	Result of applying the "Rules for result data types" on page 115 to the expressions following the WHEN keyword that are other than untyped parameter markers
At least one of the result-expressions in a CASE expression (both Simple and Searched) with the rest of the result-expressions either untyped parameter marker or NULL.	Error
Any or all expressions following WHEN in a simple CASE expression.	Result of applying the "Rules for result data types" on page 115 to the expression following CASE and the expressions following WHEN that are not untyped parameter markers.
A result-expression in a CASE expression (both simple and searched) where at least one result-expression is not NULL and not an untyped parameter marker.	Result of applying the "Rules for result data types" on page 115 to all result-expressions that are other than NULL or untyped parameter markers.
Alone as a *column-expression* in a single-row VALUES clause that is not within an INSERT statement and not within the VALUES clause of in insert operation of a MERGE statement.	Error
Alone as a *column-expression* in a multi-row VALUES clause that is not within an INSERT statement, and for which the *column-expression*s in the same position in all other *row-expressions* are untyped parameter markers.	Error
Alone as a *column-expression* in a multi-row VALUES clause that is not within an INSERT statement, and for which a *column-expression* in the same position of at least one other *row-expression* is not an untyped parameter marker or NULL.	Result of applying the "Rules for result data types" on page 115 to all operands that are other than untyped parameter markers.

Table 105. Untyped Parameter Marker Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

Untyped Parameter Marker Location	Data Type
Alone as a *column-expression* in a single-row VALUES clause within an INSERT statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. [111]
Alone as a *column-expression* in a multi-row VALUES clause within an INSERT statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. [111]
Alone as a *column-expression* in a VALUES clause of the source-table for a MERGE statement	Error
Alone as a *column-expression* in the VALUES clause of an insert operation of a MERGE statement	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. [111]
Alone as a *column-expression* on the right side of assignment-clause for an update operation of a MERGE statement	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. [111]
Alone as a value on the right hand side of a SET clause of an UPDATE statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. [111]
As a value in an *insert-multiple-rows* of an INSERT statement.	INTEGER
As a value on the right side of a SET special register statement	The data type of the special register.
As a value in the VALUES clause of the VALUES INTO statement, where the associated expression is a global variable.	The data type of the global variable.
As a value in the INTO clause of the VALUES INTO statement	The data type of the associated expression. [111]
As a value in a FREE LOCATOR or HOLD LOCATOR statement	Locator.
As a value for the password in a SET ENCRYPTION PASSWORD statement	VARCHAR(128)
As a value for the hint in a SET ENCRYPTION PASSWORD statement	VARCHAR(32)

Table 106. Untyped Parameter Marker Usage in Predicates

Untyped Parameter Marker Location	Data Type
Both operands of a comparison operator or DISTINCT predicate	VARGRAPHIC(16370) CCSID 1200
One operand of a comparison operator or DISTINCT predicate where the other operand is other than an untyped parameter marker or a distinct type.	The data type of the other operand.[111]
One operand of a comparison operator where the other operand is a distinct type.	Error

Table 106. Untyped Parameter Marker Usage in Predicates (continued)

Untyped Parameter Marker Location	Data Type
All operands of a BETWEEN predicate	VARGRAPHIC(16370) CCSID 1200
Two operands of a BETWEEN predicate	Same as that of the only non-parameter marker.
Only one operand of a BETWEEN predicate	Result of applying the "Rules for result data types" on page 115 on all operands that are other than untyped parameter markers, except the CCSID attribute is the CCSID of the value specified at execution time.
All operands of an IN predicate, for example, ? IN (?,?,?)	VARGRAPHIC(16370) CCSID 1200
The first operand of an IN predicate where the right hand side is a fullselect, for example, ? IN (fullselect).	Data type of the selected column
The first operand of an IN predicate where the right hand side is not a fullselect, for example, ? IN (?,A,B) or for example, ? IN (A,?,B,?).	Result of applying the "Rules for result data types" on page 115 on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers, except the CCSID attribute is the CCSID of the value specified at execution time.
Any or all operands of the IN list of the IN predicate, for example, for example, A IN (?,B,?).	Result of applying the "Rules for result data types" on page 115 on all operands of the IN predicate (operands to the left and right of the IN predicate) that are other than untyped parameter markers, except the CCSID attribute is the CCSID of the value specified at execution time.
Any operands in a *row-value-expression* of an IN predicate, for example, (c1,?) IN ...	Error
Any select list items in a subquery if a *row-value-expression* is specified in an IN predicate, for example, (c1,c2) IN (SELECT ?, c1 FROM ...)	Error
All three operands of the LIKE predicate.	Match expression (operand 1) and pattern expression (operand 2) VARCHAR(32740); escape expression (operand 3) is VARCHAR(1) [112] with the CCSID of the job.
The match expression of the LIKE predicate when either the pattern expression or the escape expression is other than an untyped parameter marker.	Either VARCHAR(32740) or VARGRAPHIC(16370) or VARBINARY(32740) depending on the data type of the first operand that is not an untyped parameter marker. The CCSID depends on the CCSID of the first operand.
The pattern expression of the LIKE predicate when either the match expression or the escape expression is other than an untyped parameter marker.	Either VARCHAR(32740) or VARGRAPHIC(16370) or VARBINARY(32740) depending on the data type of the first operand that is not an untyped parameter marker. The CCSID depends on the CCSID of the first operand. For information about using fixed-length variables for the value of the pattern, see "LIKE predicate" on page 213.

Table 106. Untyped Parameter Marker Usage in Predicates (continued)

Untyped Parameter Marker Location	Data Type
The escape expression of the LIKE predicate when either the match expression or the pattern expression is other than an untyped parameter marker.	Either VARCHAR(1) [112] or VARGRAPHIC(1) or VARBINARY(1) depending on the result of applying the "Rules for result data types" on page 115 on all operands that are other than untyped parameter markers. The CCSID also depends on result of applying these rules.
Operand of the NULL predicate	VARGRAPHIC(16370) CCSID 1200

Table 107. Untyped Parameter Marker Usage in Built-in Functions

Untyped Parameter Marker Location	Data Type
All arguments of BITAND, BITANDNOT, BITOR, BITXOR, BITNOT, COALESCE, IFNULL, LAND, LOR, MIN, MAX, NULLIF, VALUE, or XOR	Error
Any argument of COALESCE, IFNULL, LAND, LOR, MIN, MAX, NULLIF, or VALUE, or XOR where at least one argument is other than an untyped parameter marker.	Result of applying the "Rules for result data types" on page 115 on all arguments that are other than untyped parameter markers. If the result is a distinct type, an error is returned.
An argument of BITAND, BITANDNOT, BITOR, and BITXOR where the other argument is other than an untyped parameter marker.	If the other argument is SMALLINT, INTEGER, or BIGINT, the data type of the other argument. Otherwise, DECFLOAT(34).
All arguments of COMPARE_DECFLOAT, DECFLOAT_SORTKEY, NORMALIZE_DECFLOAT, QUANTIZE, and TOTALORDER	DECFLOAT(34)
Both arguments of LOCATE, POSITION, or POSSTR	DBCLOB(1G) CCSID 1200
One argument of LOCATE, POSITION, or POSSTR when the other argument is a character data type.	VARCHAR(32740) with the CCSID of the other argument
One argument of LOCATE, POSITION, or POSSTR when the other argument is a graphic data type.	VARGRAPHIC(16370) with the CCSID of the other argument
One argument of LOCATE, POSITION, or POSSTR when the other argument is a binary data type.	VARBINARY(32740)
Both the first and second arguments of LOCATE_IN_STRING or OVERLAY	DBCLOB(1G) CCSID 1200
The first or second argument of LOCATE_IN_STRING or OVERLAY when the other string argument is a character data type.	VARCHAR(32740) with the CCSID of the other argument
The first or second argument of LOCATE_IN_STRING or OVERLAY when the other string argument is a graphic data type.	VARGRAPHIC(16370) with the CCSID of the other argument

Table 107. Untyped Parameter Marker Usage in Built-in Functions (continued)

Untyped Parameter Marker Location	Data Type
The first or second argument of LOCATE_IN_STRING or OVERLAY when the other string argument is a binary data type.	VARBINARY(32740)
The third or fourth argument of LOCATE_IN_STRING or OVERLAY	INTEGER
The second argument of LPAD or RPAD	INTEGER
The third argument of LPAD or RPAD	VARCHAR(32740)
The argument of UPPER, LOWER, UCASE, and LCASE	DBCLOB(1G) CCSID 1200
First or second argument of MONTHS_BETWEEN	TIMESTAMP
SUBSTR (first argument)	DBCLOB(1G) CCSID 1200
SUBSTR (second and third arguments)	INTEGER
First operand of REGEXP_LIKE, REGEXP_INSTR, REGEXP_SUBSTR, REGEXP_COUNT, and REGEXP_REPLACE	DBCLOB(1G) CCSID 1200
Second operand of REGEXP_LIKE, REGEXP_INSTR, REGEXP_SUBSTR, REGEXP_COUNT, and REGEXP_REPLACE	DBCLOB(32K) CCSID 1200
Third operand of REGEXP_REPLACE	DBCLOB(32K) CCSID 1200
source-string operand of REGEXP_LIKE, REGEXP_COUNT, REGEXP_INSTR, REGEXP_SUBSTR, and REGEXP_REPLACE	DBCLOB(32K) CCSID 1200
pattern-expression of REGEXP_LIKE, REGEXP_COUNT, REGEXP_INSTR, REGEXP_SUBSTR, and REGEXP_REPLACE	DBCLOB(32K) CCSID 1200
replacement-string operand of REGEXP_REPLACE	DBCLOB(32K) CCSID 1200
start operand of REGEXP_LIKE, REGEXP_COUNT, REGEXP_INSTR, REGEXP_SUBSTR, and REGEXP_REPLACE	INTEGER
flags operand of REGEXP_LIKE, REGEXP_COUNT, REGEXP_INSTR, REGEXP_SUBSTR, and REGEXP_REPLACE	VARCHAR(6)
occurrence operand of REGEXP_INSTR, REGEXP_SUBSTR, and REGEXP_REPLACE	INTEGER
return-option operand of REGEXP_INSTR	INTEGER
group operand of REGEXP_INSTR and REGEXP_SUBSTR	INTEGER
The first argument of TRANSLATE	Error
The second and third arguments of TRANSLATE	VARCHAR(32740) if the first argument is a character type; VARGRAPHIC(16370) if the first argument is a graphic type. The CCSID depends on the CCSID of the first argument.

Table 107. Untyped Parameter Marker Usage in Built-in Functions (continued)

Untyped Parameter Marker Location	Data Type
The fourth argument of TRANSLATE	VARCHAR(1) if the first argument is a character type; VARGRAPHIC(1) if the first argument is a graphic type. The CCSID depends on the CCSID of the first argument.
The first argument of TIMESTAMP or TIMESTAMP_ISO	Error
The second argument of TIMESTAMP	TIME
The first argument of VARCHAR_FORMAT	TIMESTAMP
The first argument of TIMESTAMP_FORMAT	VARGRAPHIC(16370) CCSID 1200
The second argument of TIMESTAMP_FORMAT or VARCHAR_FORMAT	Error
First argument of XMLVALIDATE	XML [113]
First argument of XMLPARSE	CLOB(2G) or DBCLOB(1G) based on the CCSID value for the query option SQL_XML_DATA_CCSID
First argument of XMLCOMMENT	VARCHAR(32740) or VARGRAPHIC(16370) based on the CCSID value for the query option SQL_XML_DATA_CCSID
First argument of XMLTEXT	VARCHAR(32740) or VARGRAPHIC(16370) based on the CCSID value for the query option SQL_XML_DATA_CCSID
Second argument of XMLPI	VARCHAR(36740) or VARGRAPHIC(16370) based on the CCSID value for the query option SQL_XML_DATA_CCSID
First argument of XMLSERIALIZE	XML [114]
All arguments of XMLDOCUMENT	XML [113]
All arguments of XMLCONCAT	XML [113]
First, second, and third arguments of XSLTRANSFORM	XML [114]
Subindex of an ARRAY	BIGINT
Unary minus	DECFLOAT(34)
Unary plus	DECFLOAT(34)
All other arguments of all other scalar functions.	Error
Argument of an aggregate function	Error

Table 108. Untyped Parameter Marker Usage in User-defined Routines

Untyped Parameter Marker Location	Data Type
Argument of a function	Error
Argument of a procedure	The data type of the parameter, as defined when the procedure was created

111. If the data type is DATE, TIME, or TIMESTAMP, then VARCHAR(32740) is used.

112. If the escape expression is MIXED data, the data type is VARCHAR(4).

Error checking: When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is not valid, a prepared statement is not created and an error is returned.

In local and remote processing, the DLYPREP(*YES) option can cause some SQL statements to receive "delayed" errors. For example, DESCRIBE, EXECUTE, and OPEN might receive an SQLCODE that normally occurs during PREPARE processing.

Reference and execution rules: Prepared statements can be referred to in the following kinds of statements, with the following restrictions shown:

Statement	The prepared statement restrictions
DESCRIBE	None
DECLARE CURSOR	Must be SELECT when the cursor is opened
EXECUTE	Must not be SELECT

A prepared statement can be executed many times. If a prepared statement is not executed more than once and does not contain parameter markers, it is more efficient to use the EXECUTE IMMEDIATE statement rather than the PREPARE and EXECUTE statements.

Extended indicator usage: The EXTENDED INDICATORS clause indicates whether extended indicator variable values are enabled in the SET *assignment-clause* of an UPDATE statement, the VALUES *expression-list* of an INSERT statement, or the insert operation or update operation of a MERGE statement.

Extended indicator variables and deferred error checks: When extended indicator variables are enabled, the UNASSIGNED indicator variable value effectively causes its target column to be omitted from the statement. Because of this, validation that is normally done during statement preparation is delayed until statement execution.

Prepared statement persistence: All prepared statements are destroyed when:[115]

- A CONNECT (Type 1) statement is executed.
- A DISCONNECT statement disconnects the connection with which the prepared statement is associated.
- A prepared statement is associated with a release-pending connection and a successful commit occurs.
- The associated scope (job, activation group, or program) of the SQL statement ends.

Scope of a statement: The scope of *statement-name* is the source program in which it is defined. You can only reference a prepared statement by other SQL statements that are precompiled with the PREPARE statement. For example, a program called from another separately compiled program cannot use a prepared statement that was created by the calling program.

113. The CCSID for XML is determined as described in "XML Values" on page 88.

114. The CCSID is determined based on the attributes of the *data-type* specified on the AS clause as described in "CAST specification" on page 185. If the *data-type* is a binary string or bit data, then the SQL_XML_DATA_CCSID is used for the CCSID attribute.

115. Prepared statements may be cached and not actually destroyed. However, a cached statement can only be used if the same statement is prepared again.

The scope of statement-name is also limited to the thread in which the program that contains the statement is running. For example, if the same program is running in two separate threads in the same job, the second thread cannot use a statement that was prepared by the first thread.

Although the scope of a statement is the program in which it is defined, each package created from the program includes a separate instance of the prepared statement and more than one prepared statement can exist at run time. For example, assume a program using CONNECT (Type 2) statements connects to location X and location Y in the following sequence:

```
EXEC SQL CONNECT TO X;
EXEC SQL PREPARE S FROM :hv1;
EXEC SQL EXECUTE S;
.
.
.
EXEC SQL CONNECT TO Y;
EXEC SQL PREPARE S FROM :hv1;
EXEC SQL EXECUTE S;
```

The second prepare of S prepares another instance of S at Y.

A prepared statement can only be referenced in the same instance of the program in the program stack, unless CLOSQLCSR(*ENDJOB), CLOSQLCSR(*ENDACTGRP), or CLOSQLCSR(*ENDSQL) is specified on the CRTSQLxxx commands.

- If CLOSQLCSR(*ENDJOB) is specified, the prepared statement can be referred to by any instance of the program (that prepared the statement) on the program stack. In this case, the prepared statement is destroyed at the end of the job.
- If CLOSQLCSR(*ENDSQL) is specified, the prepared statement can be referred to by any instance of the program (that prepared the statement) on the program stack until the last SQL program on the program stack ends. In this case, the prepared statement is destroyed when the last SQL program on the program stack ends.
- If CLOSQLCSR(*ENDACTGRP) is specified, the prepared statement can be referred to by all instances of the module in the program that prepared the statement until the activation group ends. In this case, the prepared statement is destroyed when the activation group ends.

Allocating the SQL descriptor: If a USING clause is specified, before the PREPARE statement is executed, the SQL descriptor must be allocated using the ALLOCATE DESCRIPTOR statement. If the number of descriptor items allocated is less than the number of result columns, a warning (SQLSTATE 01005) is returned.

PREPARE and *LIBL: Normally, any unqualified names of objects are resolved when a statement is prepared. Hence, any changes to the CURRENT SCHEMA or CURRENT PATH after the statement has been prepared have no effect on which objects will be referenced when the statement is executed or opened. However, if system naming is used and an object name is implicitly qualified with *LIBL, the object is resolved at execute or open time. Any changes to the library list after the statement is prepared but before execute or open time will affect which objects will be referenced when the statement is executed or opened.

Examples

Example 1: Prepare and execute a non-*select-statement* in a COBOL program. Assume the statement is contained in a variable HOLDER and that the program will place a statement string into the variable based on some instructions from the user. The statement to be prepared does not have any parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER  END-EXEC.

EXEC SQL  EXECUTE STMT_NAME  END-EXEC.
```

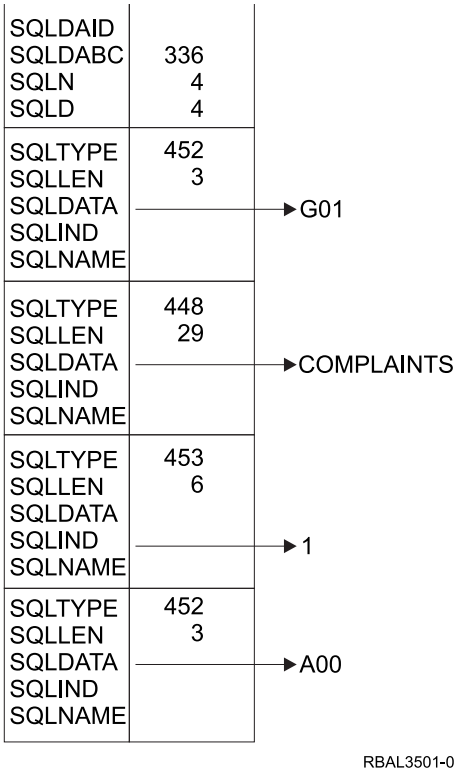
Example 2: Prepare and execute a non-*select-statement* as in example 1, except assume the statement to be prepared can contain any number of parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER  END-EXEC.

EXEC SQL  EXECUTE STMT_NAME USING DESCRIPTOR :INSERT_DA  END-EXEC.
```

Assume that the following statement is to be prepared:

```
INSERT INTO DEPARTMENT VALUES(?, ?, ?, ?)
```

To insert department number G01 named COMPLAINTS, which has no manager and reports to department A00, the structure INSERT_DA should have the following values before executing the EXECUTE statement.



REFRESH TABLE

The REFRESH TABLE statement refreshes the data in a materialized query table. The statement deletes all rows in the materialized query table and then inserts the result rows from the *select-statement* specified in the definition of the materialized query table.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

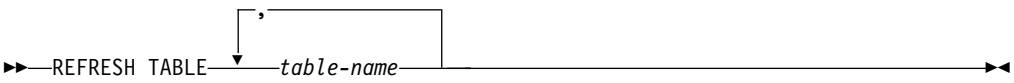
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table identified in the statement,
  - The system authority *OBJMGT on the table
  - The DELETE privilege on the table
  - The INSERT privilege on the table
  - The system authority *EXECUTE on the library containing the table
- Administrative authority

For information on the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

Syntax



Description

table-name
Identifies the materialized table to be refreshed. The *table-name* must identify a materialized query table that exists at the current server. REFRESH TABLE evaluates the *select-statement* in the definition of the materialized query table to refresh the table.

Notes

Refresh use of materialized query tables: No materialized query tables are used to evaluate the *select-statement* during the processing of REFRESH TABLE statement.

Refresh isolation level: The isolation level used to evaluate the *select-statement* is either:

- the isolation level specified on the *isolation-level* clause of the *select-statement*, or
- if the *isolation-level* clause was not specified, the isolation level of the materialized query table recorded when CREATE TABLE or ALTER TABLE was issued.

Number of rows: After successful execution of a REFRESH TABLE statement, the ROW_COUNT statement information item in the SQL Diagnostics Area (or SQLERRD(3) in the SQLCA) will contain the number of rows inserted into the materialized query table.

Example

Refresh the data in the TRANSCOUNT materialized query table.

```
REFRESH TABLE TRANSCOUNT
```

RELEASE (Connection)

The RELEASE statement places one or more connections in the release-pending state.

Invocation

This statement can only be embedded within an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

RELEASE is not allowed in a trigger or function.

Authorization

If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax

Description

server-name **or** *variable*
Identifies a connection by the specified server name or the server name contained in the variable. It can be a global variable if it is qualified with schema name. If a variable is specified:

- It must be a character-string variable.
- It must not be followed by an indicator variable.
- The server name must be left-justified within the variable and must conform to the rules for forming an ordinary identifier.
- If the length of the server name is less than the length of the variable, it must be padded on the right with blanks.

When the RELEASE statement is executed, the specified server name or the server name contained in the variable must identify an existing connection of the activation group.

CURRENT
Identifies the current connection of the activation group. The activation group must be in the connected state.

ALL or ALL SQL
Identifies all existing connections of the activation group (local as well as remote connections).

An error or warning does not occur if no connections exist when the statement is executed.

An application server named ALL can only be identified by a variable or a delimited identifier.

If the RELEASE statement is successful, each identified connection is placed in the release-pending state and will therefore be ended during the next commit operation. If the RELEASE statement is unsuccessful, the connection state of the activation group and the states of its connections are unchanged.

Notes

RELEASE and CONNECT (Type 1): Using CONNECT (Type 1) semantics does not prevent using RELEASE.

Scope of RELEASE: RELEASE does not close cursors, does not release any resources, and does not prevent further use of the connection.

Resource considerations for remote connections: Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be in the release-pending state and one that is going to be reused should not be in the release-pending state.

Connection states: ROLLBACK does not reset the state of a connection from release-pending to held.

If the current connection is in the release-pending state when a commit operation is performed, the connection is ended and the activation group is in the unconnected state. In this case, the next executed SQL statement must be CONNECT or SET CONNECTION.

RELEASE ALL places the connection to the local release-pending in the release-pending state. A connection in the release-pending state is ended during a commit operation even though it has an open cursor defined with the WITH HOLD clause.

Examples

Example 1: The connection to TOROLAB1 is not needed in the next unit of work. The following statement will cause it to be ended during the next commit operation.

```
EXEC SQL RELEASE TOROLAB1;
```

Example 2: The current connection is not needed in the next unit of work. The following statement will cause it to be ended during the next commit operation.

```
EXEC SQL RELEASE CURRENT;
```

Example 3: None of the existing connections are needed in the next unit of work. The following statement will cause it to be ended during the next commit operation.

```
EXEC SQL RELEASE ALL;
```

RELEASE SAVEPOINT

The RELEASE SAVEPOINT statement releases the identified savepoint and any subsequently established savepoints within a unit of work at the current server.

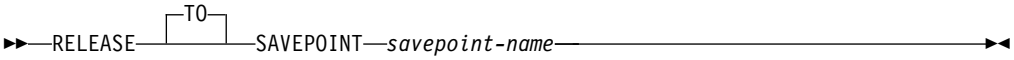
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

savepoint-name
Identifies the savepoint to release. The name must identify a savepoint that exists at the current server. The named savepoint and all the savepoints at the current server that were subsequently established in the unit of work are released. After a savepoint is released, it is no longer maintained, and rollback to the savepoint is no longer possible.

Notes

Savepoint Names: The name of the savepoint that was released can be re-used in another SAVEPOINT statement, regardless of whether the UNIQUE keyword was specified on an earlier SAVEPOINT statement specifying this same savepoint name.

Isolation Level Restriction: A RELEASE SAVEPOINT statement is not allowed if commitment control is not active for the activation group. For information about determining which commitment definition is used, see "Notes" on page 824 in COMMIT statement.

Example

Assume that a main routine sets savepoint A and then invokes a subroutine that sets savepoints B and C. When control returns to the main routine, release savepoint A and any subsequently set savepoints. Savepoints B and C, which were set by the subroutine, are released in addition to A.

```
RELEASE SAVEPOINT A
```

RENAME

The RENAME statement renames a table, view, or index. The name and/or the system object name of the table, view, or index can be changed.

Invocation

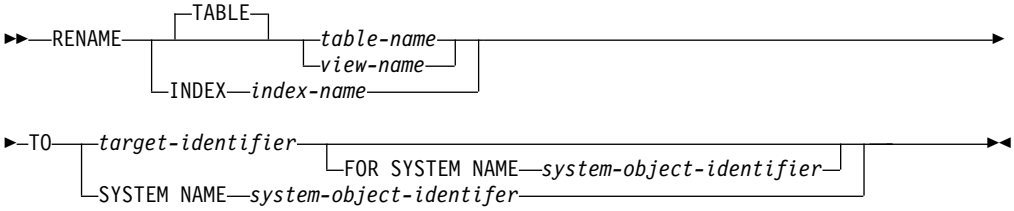
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
  - If the name of the object is changed:
    - The system authority of *OBJMGT on the table, view, or index to be renamed
    - The system authority *EXECUTE on the library containing the table, view, or index to be renamed
  - If the system name of the object is changed:
    - The system authority of *OBJMGT on the table, view, or index to be renamed
    - The system authorities *EXECUTE and *UPD on the library containing the table, view, or index to be renamed
- Administrative authority

Syntax



Description

TABLE *table-name* **or** *view-name*

Identifies the table or view that will be renamed. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a catalog table or a declared temporary table. The specified name can be an alias name. The specified table or view is renamed to the new name. All privileges, constraints, indexes, triggers, views, and logical files on the table or view are preserved.

Any access plans that reference the table or view are implicitly prepared again when a program that uses the access plan is next run. Since the program refers to a table or view with the original name, if a table or view with the original name does not exist at that time, an error is returned.

INDEX *index-name*
Identifies the index that will be renamed. The *index-name* must identify an index that exists at the current server. The specified index is renamed to the new name.

Any access plans that reference the index are not affected by rename.

target-identifier
Identifies the new *table-name*, *view-name*, or *index-name* of the table, view, or index, respectively. *target-identifier* must not be the same as a table, view, alias, or index that already exists at the current server. The *target-identifier* must be an unqualified SQL identifier.

SYSTEM NAME *system-object-identifier*
Identifies the new *system-object-identifier* of the table, view, or index, respectively. *system-object-identifier* must not be the same as a table, view, alias, or index that already exists at the current server. The *system-object-identifier* must be an unqualified system identifier.

If the name of the object and the system name of the object are the same and *target-identifier* is not specified, specifying *system-object-identifier* will be the new name and system object name. Otherwise, specifying *system-object-identifier* will only affect the system name of the object and not affect the name of the object.

If both *target-identifier* and *system-object-identifier* are specified, they cannot both be valid system object names.

Notes

Effects of the statement: The specified table, view, or index is renamed to the new name. For a renamed table, all privileges and indexes on the table are preserved. For a renamed index, all privileges are preserved.

Invalidation of packages and access plans: Any access plans that refer to that table are invalidated. For more information see "Packages and access plans" on page 15.

Considerations for aliases: If an alias name is specified for *table-name*, the table must exist at the current server, and the table that is identified by the alias is renamed. The name of the alias is not changed and continues to refer to the old table name after the rename.

There is no support for changing the name of an alias with the RENAME statement. To change the name to which the alias refers, the alias must be dropped and recreated.

Rename rules: The rename operation performed depends on the new name specified.

- If the new name is a valid system identifier,
  - the alternative name (if any) is removed, and
  - the system object name is changed to the new name.
- If the new name is not a valid system identifier,
  - the alternative name is added or changed to the new name, and
  - a new system object name is generated if the system object name (of the table or view) was specified as the table, view, or index to rename. For more information about generated table name rules, see "Rules for Table Name Generation" on page 1030.

If an alias name is specified for *table-name*, the alias must exist at the current server, and the table that is identified by the alias is renamed. The name of the alias is not changed and continues to refer to the old table after the rename. There is no support for changing the name of an alias.

Examples

Example 1: Change the name of the EMPLOYEE table to CURRENT_EMPLOYEES:

```
RENAME TABLE EMPLOYEE
TO CURRENT_EMPLOYEES
```

Example 2: Change the name of the unique index using EMPNO, called XEMP1, to UXEMPNO:

```
RENAME INDEX XEMP1
  TO UXEMPNO
```

Example 3: Rename a table named MY_IN_TRAY to MY_IN_TRAY_94. The system object name will remain unchanged (MY_IN_TRAY).

```
RENAME TABLE MY_IN_TRAY TO MY_IN_TRAY_94
  FOR SYSTEM NAME MY_IN_TRAY
```

REVOKE (Function or Procedure Privileges)

This form of the REVOKE statement removes the privileges on a function or procedure.

Invocation

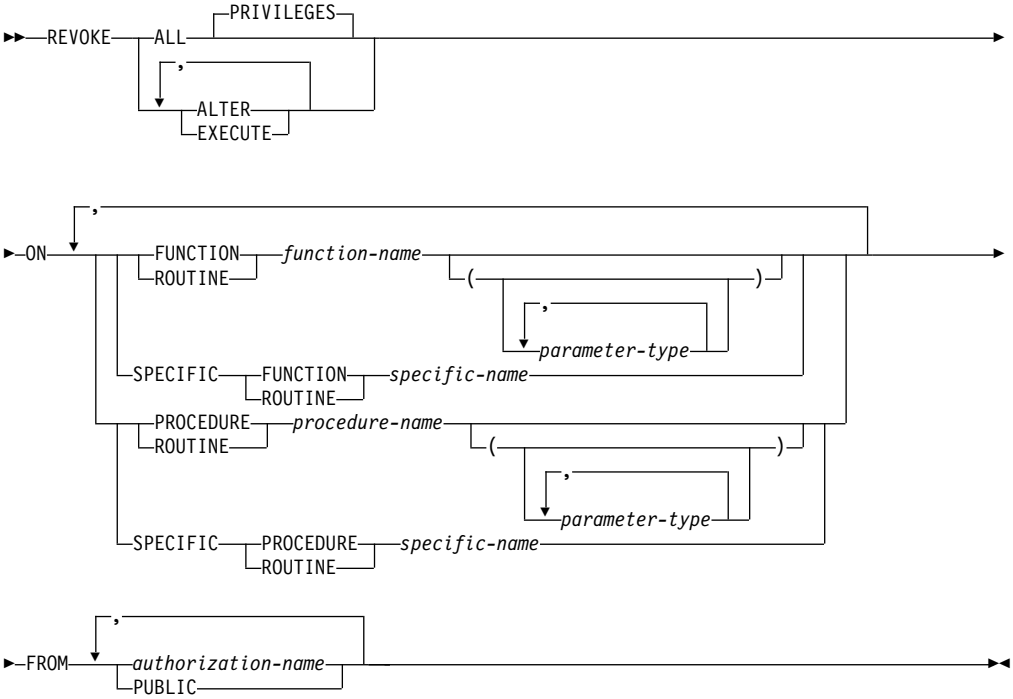
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

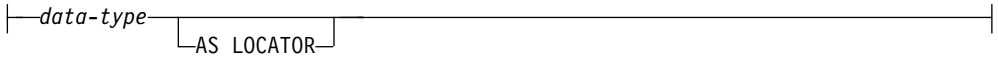
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each function or procedure identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the function or procedure
  - The system authority *EXECUTE on the library (or directory if this is a Java routine) containing the function or procedure
- Administrative authority

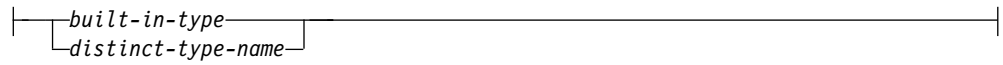
Syntax



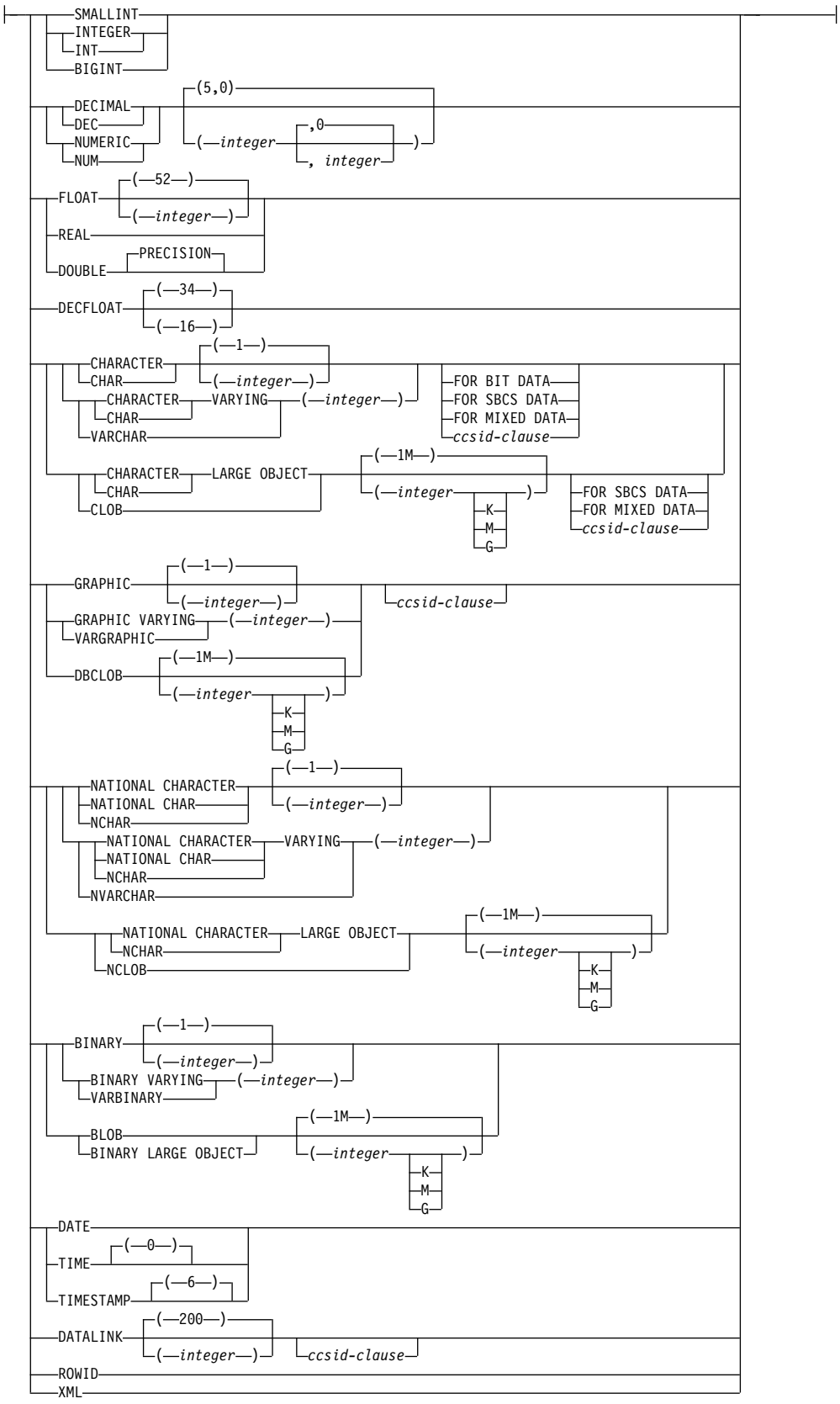
parameter-type:



data-type:



built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(integer
,0
, integer
)
FLOAT
(52)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
FOR BIT DATA
FOR SBCS DATA
FOR MIXED DATA
ccsid-clause
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
K
M
G
GRAPHIC
GRAPHIC VARYING
VARGRAPHIC
DBCLOB
NATIONAL CHARACTER
NATIONAL CHAR
NCHAR
NATIONAL CHARACTER VARYING
NATIONAL CHAR VARYING
NVARCHAR
NATIONAL CHARACTER LARGE OBJECT
NCHAR LARGE OBJECT
NCLOB
BINARY
BINARY VARYING
VARBINARY
BLOB
BINARY LARGE OBJECT
DATE
TIME
(0)
TIMESTAMP
(6)
DATALINK
(200)
ROWID
XML

ccsid-clause:

Description

ALL or ALL PRIVILEGES
Revokes one or more function or procedure privileges from each *authorization-name*. The privileges revoked are those privileges on the identified functions or procedures that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a function or procedure is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER
Revokes the privilege to use the ALTER FUNCTION, ALTER PROCEDURE, or COMMENT statement.

EXECUTE
Revokes the privilege to execute a function or procedure.

FUNCTION or SPECIFIC FUNCTION
Identifies the function from which the privilege is revoked. The function must exist at the current server and it must be a user-defined function, but not a function that was implicitly generated with the creation of a distinct type. The function can be identified by its name, function signature, or specific name.

FUNCTION *function-name*
Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*
Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which the privilege is to be revoked. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name
Identifies the name of the function.

(parameter-type, ...)
Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For

example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).

- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE or SPECIFIC PROCEDURE
Identifies the procedure from which the privilege is revoked. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*
Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be revoked. Synonyms for data types are considered a match. Parameters that have defaults must be included in this signature.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name
Identifies the name of the procedure.

(parameter-type, ...)
Identifies the parameters of the procedure.

If an unqualified distinct type or array type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type or array type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR
Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or XML or a distinct type based on a LOB or XML.

SPECIFIC PROCEDURE *specific-name*
Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

FROM
Identifies from whom the privileges are revoked.

authorization-name,...
Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 18.

Notes

Multiple grants: If you revoke a privilege on a function or procedure, it nullifies any grant of the privilege on that function or procedure, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding system authorities: When a function or procedure privilege is revoked, the corresponding system authorities are revoked. For information about the system authorities that correspond to SQL privileges see "GRANT (Function or Procedure Privileges)" on page 1219.

Privileges revoked from either an SQL or external function or procedure are revoked from its associated program (*PGM) or service program (*SRVPGM) object. Privileges revoked from a Java external function or procedure are revoked from the associated class file or jar file. If the associated program, service program, class file, or jar file is not found when the revoke is executed, an error is returned.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword RUN can be used as a synonym for EXECUTE.

Example

Revoke the EXECUTE privilege on procedure PROCA from PUBLIC.

```
REVOKE EXECUTE
  ON PROCEDURE PROCA
  FROM PUBLIC
```

REVOKE (Global Variable Privileges)

This form of the REVOKE statement removes the privileges on a created global variable.

Invocation

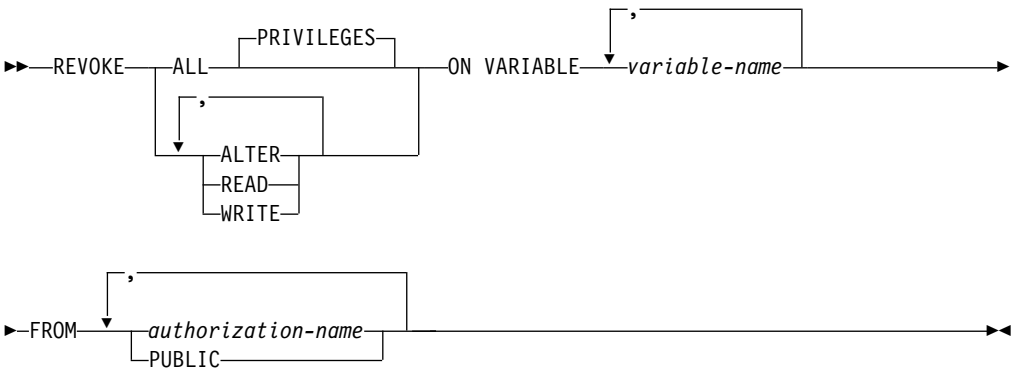
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each variable identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the global variable
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Revokes one or more global variable privileges from each *authorization-name*. The privileges revoked are those privileges on the identified global variables that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a global variable is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER
Revokes the privilege to use COMMENT and LABEL statements on the specified global variables.

READ
Revokes the privilege to read the value of the specified global variables.

WRITE
Revokes the privilege to assign a value to the specified global variables.

ON VARIABLE *variable-name*
Identifies the global variables from which one or more privileges are to be revoked. Each *variable-name* must identify a global variable that exists at the current server.

FROM
Identifies from whom the privileges are revoked.

authorization-name,...
Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 18.

Notes

Multiple grants: If you revoke a privilege on a variable, it nullifies any grant of the privilege on that variable, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding system authorities: When a global variable privilege is revoked, the corresponding system authorities are revoked. For information on the system authorities that correspond to SQL privileges see "GRANT (Global Variable Privileges)" on page 1227.

Example

REVOKE the WRITE privilege on a global variable MYSCHEMA.MYJOB_PRINTER from user ZUBIRI.

```
REVOKE WRITE
  ON VARIABLE MYSCHEMA.MYJOB_PRINTER
  FROM ZUBIRI
```

REVOKE (Package Privileges)

This form of the REVOKE statement removes the privileges on a package.

Invocation

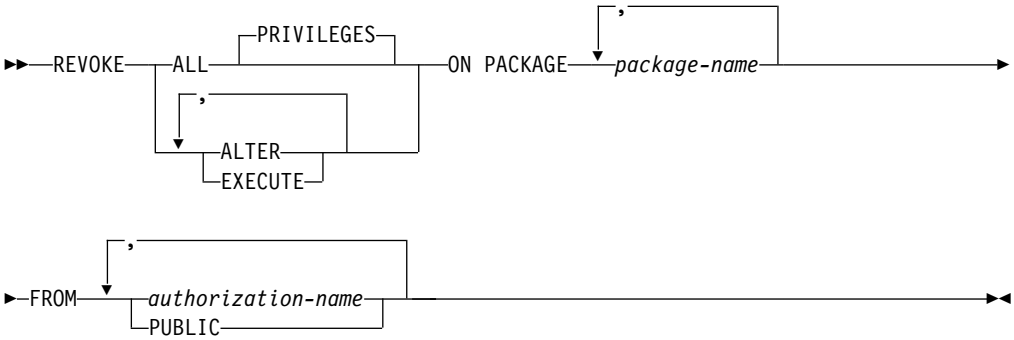
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each package identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the package
  - The system authority *EXECUTE on the library containing the package
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Revokes one or more package privileges from each *authorization-name*. The privileges revoked are those privileges on the identified packages that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a package is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER
Revokes the privilege to use the COMMENT and LABEL statements.

EXECUTE
Revokes the privilege to execute statements in a package.

ON PACKAGE *package-name*
Identifies the package from which the EXECUTE privilege is revoked. The *package-name* must identify a package that exists at the current server.

FROM
Identifies from whom the privileges are revoked.

authorization-name,...
Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 18.

Notes

Multiple grants: If you revoke a privilege on a package, it nullifies any grant of the privilege on that package, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding system authorities: When a package privilege is revoked, the corresponding system authorities are revoked. For information about the system authorities that correspond to SQL privileges see "GRANT (Package Privileges)" on page 1230.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword RUN can be used as a synonym for EXECUTE.
- The keyword PROGRAM can be used as a synonym for PACKAGE.

Example

Example 1: Revoke the EXECUTE privilege on package PKGA from PUBLIC.

```
REVOKE EXECUTE
  ON PACKAGE PKGA
  FROM PUBLIC
```

Example 2: Revoke the EXECUTE privilege on package RRSP_PKG from user FRANK and PUBLIC.

```
REVOKE EXECUTE
  ON PACKAGE RRSP_PKG
  FROM FRANK, PUBLIC
```

REVOKE (Sequence Privileges)

This form of the REVOKE statement removes the privileges on a sequence.

Invocation

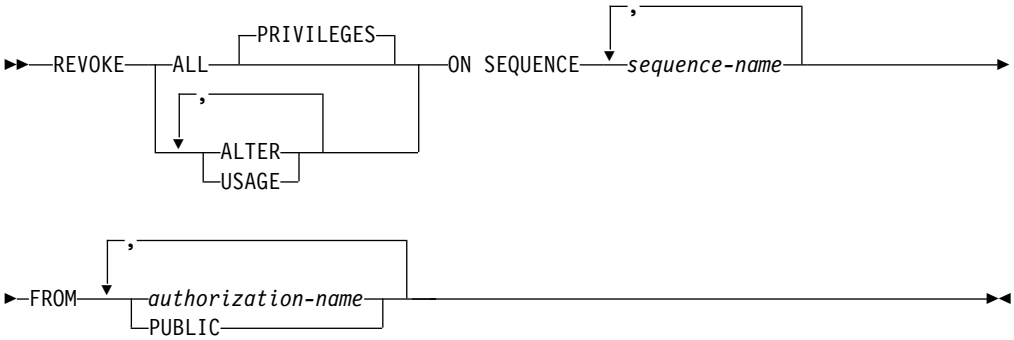
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each sequence identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the sequence
  - The system authority *EXECUTE on the library containing the sequence
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Revokes one or more sequence privileges from each *authorization-name*. The privileges revoked are those privileges on the identified sequences that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a sequence is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER
Revokes the privilege to use the ALTER SEQUENCE, COMMENT, and LABEL statements on a sequence.

USAGE
Revokes the privilege to use the sequence in NEXT VALUE or PREVIOUS VALUE expressions.

ON SEQUENCE *sequence-name*
Identifies the sequence from which the privilege is revoked. The *sequence-name* must identify a sequence that exists at the current server.

FROM
Identifies from whom the privileges are revoked.

authorization-name,...
Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 18.

Notes

Multiple grants: If you revoke a privilege on a sequence, it nullifies any grant of the privilege on that sequence, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding system authorities: When a sequence privilege is revoked, the corresponding system authorities are revoked. For information on the system authorities that correspond to SQL privileges see "GRANT (Sequence Privileges)" on page 1233.

Example

REVOKE the USAGE privilege from PUBLIC on a sequence called ORG_SEQ.

```
REVOKE USAGE
  ON SEQUENCE ORG_SEQ
  FROM PUBLIC
```

REVOKE (Table or View Privileges)

This form of the REVOKE statement removes privileges on a table or view.

Invocation

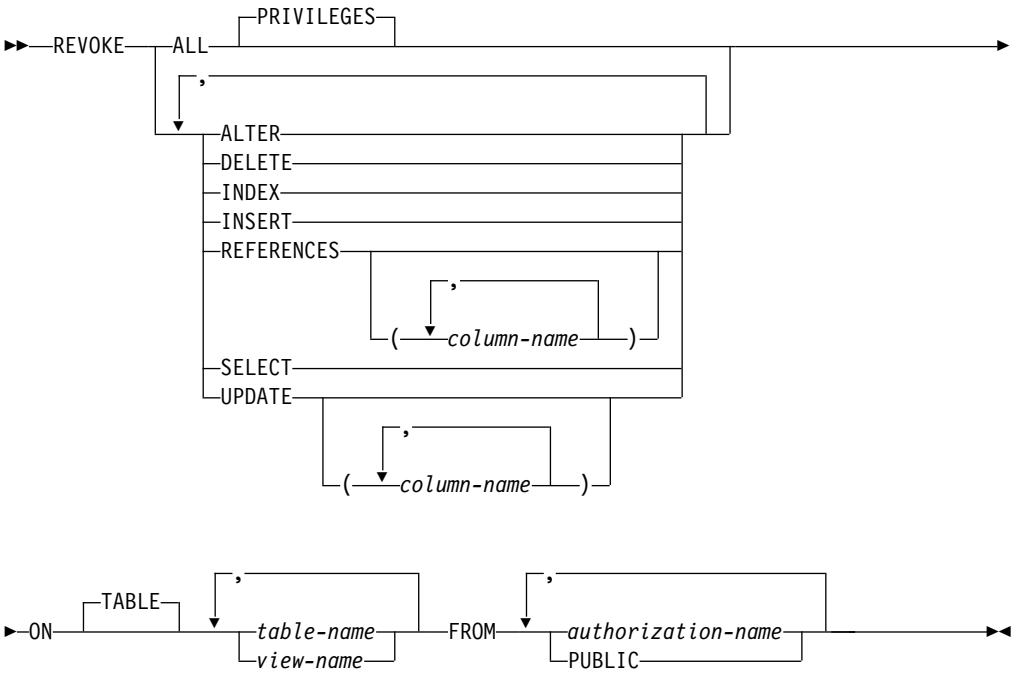
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the table or view
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Revokes one or more privileges from each *authorization-name*. The privileges revoked are those privileges on the identified tables and views that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a table or view is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described, but only as it applies to the tables and views named in the ON clause.

ALTER
Revokes the privilege to alter the specified table and the privilege to add a comment, add a label, or create an index on the specified table or view.

DELETE
Revokes the privilege to delete rows from the specified table or view.

INDEX
Revokes the privilege to create an index on the specified table.

INSERT
Revokes the privilege to insert rows in the specified table or view.

REFERENCES
Revokes the privilege to add a referential constraint in which the table is a parent.

REFERENCES (*column-name***,...)**
Revokes the privilege to add a referential constraint using the specified column(s) in the parent key. Each column name must be an unqualified name that identifies a column in each table identified in the ON clause.

SELECT
Revokes the privilege to use the SELECT or CREATE VIEW statement.

UPDATE
Revokes the privilege to use the UPDATE statement.

UPDATE (*column-name***,...)**
Revokes the privilege to update the specified columns. Each column name must be an unqualified name that identifies a column in each table identified in the ON clause.

ON *table-name* **or** *view-name*, ...
Identifies the table or view from which the privileges are revoked. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a declared temporary table.

FROM
Identifies from whom the privileges are revoked.

authorization-name,...
Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 18.

Notes

Multiple grants: If the same privilege is granted to the same user more than once, revoking that privilege from that user nullifies all those grants.

If you revoke a privilege, it nullifies any grant of that privilege, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

If more than one system authority will be revoked with an SQL privilege, and any one of the authorities cannot be revoked, then a warning occurs and no authorities will be revoked for that privilege.

Corresponding system authorities: When a table privilege is revoked, the corresponding system authorities are revoked, except:

- When revoking authorities to a table or view, *OBJOPR is revoked only when *ADD, *DLT, *READ, and *UPD have all been revoked.
- When revoking authorities to a view, authorities will not be revoked from any tables or views referenced in the fullselect of the view definition.

For information about the system authorities that correspond to SQL privileges see "GRANT (Table or View Privileges)" on page 1236.

Revoking either the INDEX or ALTER privilege, revokes the system authority *OBJALTER.

Examples

Example 1: Revoke SELECT privileges on table EMPLOYEE from user ENGLES.

```
REVOKE SELECT
  ON TABLE EMPLOYEE
  FROM ENGLES
```

Example 2: Revoke update privileges on table EMPLOYEE previously granted to all users. Note that grants to specific users are not affected.

```
REVOKE UPDATE
  ON TABLE EMPLOYEE
  FROM PUBLIC
```

Example 3: Revoke all privileges on table EMPLOYEE from users PELLOW and ANDERSON.

```
REVOKE ALL
  ON TABLE  EMPLOYEE
  FROM PELLOW, ANDERSON
```

Example 4: Revoke the privilege to update column_1 in VIEW1 from FRED.

```
REVOKE UPDATE(column_1)
  ON VIEW1
  FROM FRED
```

REVOKE (Type Privileges)

This form of the REVOKE statement removes the privileges on a type.

Invocation

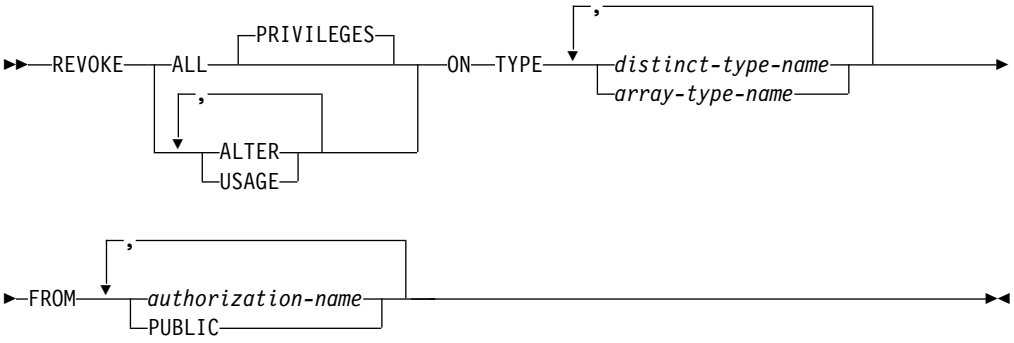
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type or array type identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the type
  - The system authority *EXECUTE on the library containing the type
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Revokes one or more type privileges from each *authorization-name*. The privileges revoked are those privileges on the identified user-defined types that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a type is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER
Revokes the privilege to use the COMMENT and LABEL statements.

USAGE
Revokes the privilege to use user-defined types in tables, functions, procedures, or as the source type in a CREATE TYPE statement.

ON TYPE *distinct-type-name* **or** *array-type-name*
Identifies the distinct type from which the privilege is revoked. The *distinct-type-name* or *array-type-name* must identify a user-defined type that exists at the current server.

FROM
Identifies from whom the privileges are revoked.

authorization-name,...
Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 18.

Notes

Multiple grants: If authorization ID A granted the same privilege to authorization ID B more than once, revoking that privilege from B nullifies all those grants.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding system authorities: When a type privilege is revoked, the corresponding system authorities are revoked. For information about the system authorities that correspond to SQL privileges see "GRANT (Type Privileges)" on page 1242.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords DATA TYPE or DISTINCT TYPE can be used as a synonym for TYPE.

Example

Revoke the USAGE privilege on distinct type SHOESIZE from user JONES.

```
REVOKE USAGE
  ON DISTINCT TYPE SHOESIZE
  FROM JONES
```

REVOKE (XML Schema Privileges)

This form of the REVOKE statement removes the privileges on an XSR object.

Invocation

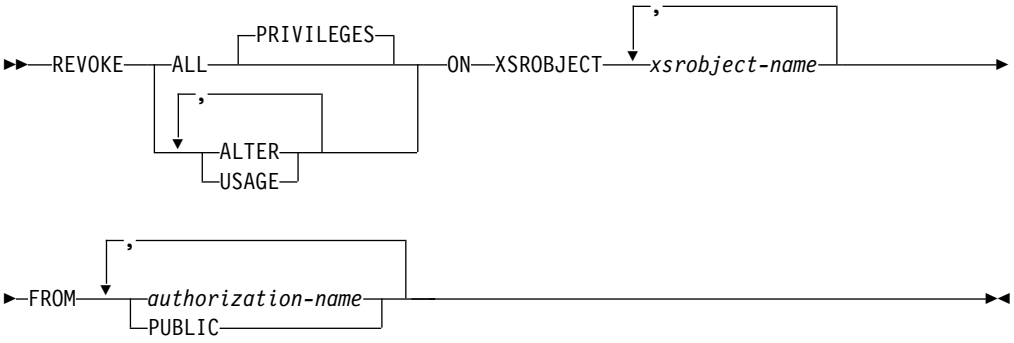
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each XSR object identified in the statement:
  - Every privilege specified in the statement
  - The system authority of *OBJMGT on the XSR object
  - The system authority *EXECUTE on the library containing the XSR object
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES
Revokes one or more XSR object privileges from each *authorization-name*. The privileges revoked are those privileges on the identified XSR objects that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on an XSR object is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER
Revokes the privilege to use the COMMENT and LABEL statements.

USAGE
Revokes the privilege to use an XSR object.

ON XSROBJECT *xsrobject-name*
Identifies the XSR objects for which the privilege is revoked. The *xsrobject-name* must identify an XSR object that exists at the current server.

FROM
Identifies from whom the privileges are revoked.

authorization-name,...
Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 18.

Notes

Multiple grants: If you revoke a privilege on an XSR object, it nullifies any grant of the privilege on that XSR object, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding system authorities: When an XSR object privilege is revoked, the corresponding system authorities are revoked. For information about the system authorities that correspond to SQL privileges see "GRANT (XML Schema Privileges)" on page 1245.

Example

Revoke the USAGE privilege on XSR object XMLSCHEMA from PUBLIC.

```
REVOKE USAGE
  ON XSRSOBJECT XMLSCHEMA
  FROM PUBLIC
```

ROLLBACK

The ROLLBACK statement is used to back out changes.

The ROLLBACK statement can be used to either:

- End a unit of work and back out all the relational database changes that were made by that unit of work. If relational databases are the only recoverable resources used by the application process, ROLLBACK also ends the unit of work.
- Back out only the changes made after a savepoint was set within the unit of work without ending the unit of work. Rolling back to a savepoint enables selected changes to be undone.

Invocation

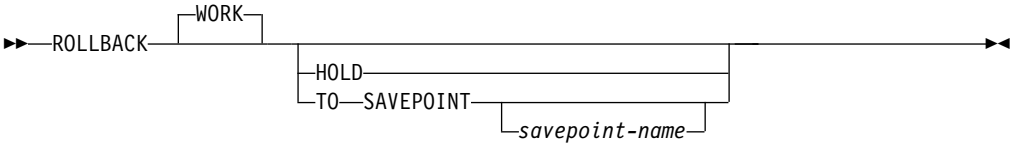
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

ROLLBACK is not allowed in a trigger if the trigger program and the triggering program run under the same commitment definition. ROLLBACK is not allowed in a procedure if the procedure is called on a Distributed Unit of Work connection to a remote application server or if the procedure is defined as ATOMIC. ROLLBACK is not allowed in a function.

Authorization

None required.

Syntax



Description

When ROLLBACK is used without the TO SAVEPOINT clause, the unit of work in which it is executed is ended. All changes made by SQL schema statements and SQL data change statements during the unit of work are backed out. For more information see Chapter 6, “Statements,” on page 699.

The generation of identity values is not under transaction control. Values generated and consumed by inserting rows into a table that has an identity column are independent of executing the ROLLBACK statement. Also, executing the ROLLBACK statement does not affect the IDENTITY_VAL_LOCAL function.

Special registers are not under transaction control. Executing a ROLLBACK statement does not affect special registers.

Sequences are not under transaction control. Executing a ROLLBACK statement does not affect the current value generated and consumed by executing a NEXT VALUE expression.

Global variables are not under transaction control. Executing a ROLLBACK statement does not affect the value of any instantiated global variable.

The impact of ROLLBACK or ROLLBACK TO SAVEPOINT on the contents of a declared temporary table is determined by the setting of the ON ROLLBACK clause of the DECLARE GLOBAL TEMPORARY TABLE statement.

WORK
ROLLBACK WORK has the same effect as ROLLBACK.

HOLD
Specifies a hold on resources. If specified, currently open cursors are not closed cursors whether they are declared with a HOLD option or not. All resources acquired during the unit of work, except locks on the rows of tables, are held. Locks on specific rows implicitly acquired during the unit of work, however, are released.

At the end of a ROLLBACK HOLD, the cursor position is the same as it was at the start of the unit of work, unless

- ALWBLK(*ALLREAD) was specified when the program or routine that contains the cursor was created
- ALWBLK(*READ) and ALWCPYDTA(*OPTIMIZE) were specified when the program or routine that contains the cursor was created

TO SAVEPOINT
Specifies that the unit of work is not to be ended and that only a partial rollback (to a savepoint) is to be performed. If a savepoint name is not specified, rollback is to the last active savepoint. For example, if in a unit of work, savepoints A, B, and C are set in that order and then C is released, ROLLBACK TO SAVEPOINT causes a rollback to savepoint B. If no savepoint is active, an error is returned.

savepoint-name
Identifies the savepoint to which to roll back. The name must identify a savepoint that exists at the current server.

After a successful ROLLBACK TO SAVEPOINT, the savepoint continues to exist.

All database changes (including changes made to declared temporary tables that were declared with the ON ROLLBACK PRESERVE ROWS clause) that were made after the savepoint was set are backed out. All locks and LOB locators are retained.

The impact on cursors resulting from a ROLLBACK TO SAVEPOINT depends on the statements within the savepoint:

- If the savepoint contains SQL schema statements on which a cursor is dependent, the cursor is closed. Attempts to use such a cursor after a ROLLBACK TO SAVEPOINT results in an error.
- Otherwise, the cursor is not affected by the ROLLBACK TO SAVEPOINT (it remains open and positioned).

Any savepoints at the current server that are set after the one to which rollback is performed are released. The savepoint to which rollback is performed is not released.

Notes

Recommended coding practices: Code an explicit COMMIT or ROLLBACK statement at the end of an application process. Either an implicit commit or rollback operation will be performed at the end of an application process depending on the application environment. Thus, a portable application should explicitly execute a COMMIT or ROLLBACK before execution ends in those environments where explicit COMMIT or ROLLBACK is permitted.

Other effects of rollback: Rollback without the TO SAVEPOINT clause and HOLD clause causes the following to occur:

- All cursors that were opened during the unit of work are closed whether they are declared with a HOLD option or not.
- All LOB locators, including those that are held, are freed.
- All locks acquired under this unit of work's commitment definition are released.

ROLLBACK has no effect on the state of connections.

Implicit ROLLBACK: The ending of the default activation group causes an implicit rollback. Thus, an explicit COMMIT or ROLLBACK statement should be issued before the end of the default activation group.

A ROLLBACK is automatically performed when:

1. The default activation group ends without a final COMMIT being issued.
2. A failure occurs that prevents the activation group from completing its work (for example, a power failure).

   If the unit of work is in the prepared state because a COMMIT was in progress when the failure occurred, a rollback is not performed. Instead, resynchronization of all the connections involved in the unit of work will occur. For more information, see the Commitment control topic collection.
3. A failure occurs that causes a loss of the connection to an application server (for example, a communications line failure).

   If the unit of work is in the prepared state because a COMMIT was in progress when the failure occurred, a rollback is not performed. Instead, resynchronization of all the connections involved in the unit of work will occur. For more information, see the Commitment control topic collection.
4. An activation group other than the default activation group ends abnormally.

Row lock limit: A unit of work may include the processing of up to and including 4 million rows, including rows retrieved during a SELECT INTO or FETCH statement, and rows inserted, deleted, or updated as part of INSERT, DELETE, and UPDATE operations.[117]

Unaffected statements: The commit and rollback operations do not affect the DROP SCHEMA statement, and this statement is not, therefore, allowed if the current isolation level is anything other than No Commit (NC).

---

116. Unless you specified COMMIT(*CHG) or COMMIT(*CS), in which case these rows are not included in this total.

117. This limit also includes:

    - Any rows accessed or changed through files opened under commitment control through high-level language file processing
    - Any rows deleted, updated, or inserted as a result of a trigger or CASCADE, SET NULL, or SET DEFAULT referential integrity delete rule.

ROLLBACK restrictions: A ROLLBACK statement is not allowed if commitment control is not active for the activation group. For information about determining which commitment definition is used, see the commitment definition discussion in the COMMIT statement.

A commit or rollback in a user-defined function in a secondary thread is not allowed.

ROLLBACK has no effect on the state of connections.

If, within a unit of work, a CLOSE is followed by a ROLLBACK, all changes made within the unit of work are backed out. The CLOSE itself is not backed out and the file is not reopened.

Examples

Example 1: See the "Example" on page 826 under COMMIT for examples using the ROLLBACK statement.

Example 2: After a unit of recovery started, assume that three savepoints A, B, and C were set and that C was released:

```
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
...
SAVEPOINT B ON ROLLBACK RETAIN CURSORS;
....
SAVEPOINT C ON ROLLBACK RETAIN CURSORS;
...
RELEASE SAVEPOINT C
```

Roll back all database changes only to savepoint A:

```
ROLLBACK WORK TO SAVEPOINT A
```

If a savepoint name was not specified (that is, ROLLBACK WORK TO SAVEPOINT), the rollback would be to the last active savepoint that was set, which is B.

SAVEPOINT

The SAVEPOINT statement sets a savepoint within a unit of work to identify a point in time within the unit of work to which relational database changes can be rolled back.

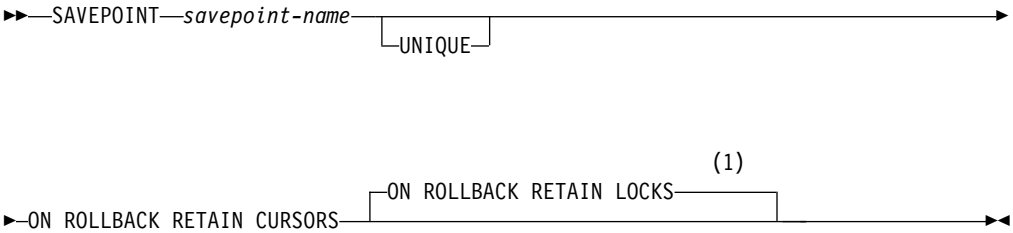
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Notes:

1 The ROLLBACK options can be specified in any order.

Description

savepoint-name
Identifies a new *savepoint*. The specified *savepoint-name* cannot begin with 'SYS'.

UNIQUE
Specifies that the application program cannot reuse the savepoint name within the unit of work. An error occurs if a savepoint with the same name as *savepoint-name* already exists within the unit of work.

Omitting UNIQUE indicates that the application can reuse the savepoint name within the unit of work. If *savepoint-name* identifies a savepoint that already exists within the unit of work and the savepoint was not created with the UNIQUE option, the existing savepoint is destroyed and a new savepoint is created. Destroying a savepoint to reuse its name for another savepoint is not the same as releasing the savepoint. Reusing a savepoint name destroys only one savepoint. Releasing a savepoint with the RELEASE SAVEPOINT statement releases the savepoint and all savepoints that have been subsequently set.

ON ROLLBACK RETAIN CURSORS
Specifies that cursors that are opened after the savepoint is set are not closed upon rollback to the savepoint.

- If SQL schema statements are executed for a table or view within the scope of the SAVEPOINT statement, any cursor that references that table or view is closed. Attempts to use such a cursor after a ROLLBACK TO SAVEPOINT results in an error.

- Otherwise, the cursor is not affected by the ROLLBACK TO SAVEPOINT (it remains open and positioned).

Although these cursors remain open after rollback to the savepoint, they might not be usable. For example, if rolling back to the savepoint causes the insertion of a row on which the cursor is positioned to be rolled back, using the cursor to update or delete the row results in an error.

ON ROLLBACK RETAIN LOCKS
Specifies that any locks that are acquired after the savepoint is set are not released on rollback to the savepoint.

Notes

Savepoint persistence: A savepoint, *S*, is destroyed when:

- A COMMIT or ROLLBACK (without a TO SAVEPOINT clause) statement is executed.
- A ROLLBACK TO SAVEPOINT statement is executed that specifies savepoint *S* or a savepoint that was established earlier than *S* in the unit of work.
- A RELEASE SAVEPOINT statement is executed that specifies savepoint *S* or a savepoint that was established earlier than *S* in the unit of work.
- A SAVEPOINT statement specifies the same name as an existing savepoint that was not created with the UNIQUE keyword.

Effect on INSERT: In an application, inserts may be buffered. The buffer will be flushed when SAVEPOINT, ROLLBACK, or RELEASE TO SAVEPOINT statements are issued.

SAVEPOINT restriction: A SAVEPOINT statement is not allowed if commitment control is not active for the activation group. For information about determining which commitment definition is used, see “Notes” on page 824 in COMMIT statement.

Example

Assume that you want to set three savepoints at various points in a unit of work. Name the first savepoint A and allow the savepoint name to be reused. Name the second savepoint B and do not allow the name to be reused. Because you no longer need savepoint A when you are ready to set the third savepoint, reuse A as the name of the savepoint.

```
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
.
.
.
SAVEPOINT B UNIQUE ON ROLLBACK RETAIN CURSORS;
.
.
.
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
```

SELECT

The SELECT statement is a form of query. It can be embedded in an SQLJ application program or issued interactively.

For detailed information, see “select-clause” on page 629 and Chapter 5, “Queries,” on page 627.

SELECT INTO

The SELECT INTO statement produces a result table consisting of at most one row, and assigns the values in that row to variables.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

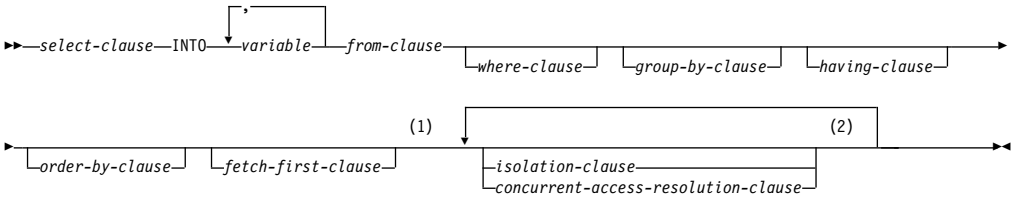
- For each table or view identified in the statement,
  - The SELECT privilege on the table or view, and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

If a global variable is specified in the INTO variable list, the privileges held by the authorization ID of the statement must include:

- The WRITE privilege on the global variable.

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

Syntax



Notes:

1 Only one row may be specified in the fetch-first-clause.

2 Each clause may be specified only once.

Description

The result table is derived by evaluating the *isolation-clause*, *concurrent-access-resolution-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*, *fetch-first-clause*, and *select-clause*, in this order.

See Chapter 5, "Queries," on page 627 for a description of the *select-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*, *fetch-first-clause*, *isolation-clause*, and *concurrent-access-resolution-clause*.

INTO *variable*,...
: Identifies one or more host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and

variables. In the operational form of the INTO clause, a reference to a host structure is replaced by a reference to each of its variables. The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on. The data type of each variable must be compatible with its corresponding column.

Notes

Variable assignment: Each assignment to a variable is performed according to the retrieval assignment rules described in "Assignments and comparisons" on page 98.[118] If the number of variables is less than the number of values in the row, an SQL warning (SQLSTATE 01503) is returned (and the SQLWARN3 field of the SQLCA is set to 'W'). Note that there is no warning if there are more variables than the number of result columns. If a value is null, an indicator variable must be provided for that value.

If the specified variable is character and is not large enough to contain the result, a warning (SQLSTATE 01004) is returned (and 'W' is assigned to SQLWARN1 in the SQLCA). The actual length of the result may be returned in the indicator variable associated with the variable, if an indicator variable is provided. For further information, see "Variables" on page 147.

If an assignment error occurs, the value of that variable and any following variables is unpredictable. Any values that have already been assigned to variables remain assigned.

Multiple assignments: If more than one variable is specified in the INTO clause, the query is completely evaluated before the assignments are performed. Thus, references to a *variable* in the select list are always the value of the *variable* prior to any assignment in the SELECT INTO statement.

Empty result table: If the result table is empty, the statement assigns '02000' to the SQLSTATE variable and does not assign values to the variables.

Result tables with more than one row: If more than one row satisfies the search condition, statement processing is terminated and an error is returned (SQLSTATE 21000). If an error occurs because the result table has more than one row, values may or may not be assigned to the variables. If values are assigned to the variables, the row that is the source of the values is undefined and not predictable.

Result column evaluation considerations: If an error occurs while evaluating a result column in the select list of a SELECT INTO statement, as the result of an arithmetic expression (such as division by zero, or overflow) or a numeric or character conversion error, the result is the null value. As in any other case of a null value, an indicator variable must be provided. The value of the variable is undefined. In this case, however, the indicator variable is set to the value of -2. Processing of the statement continues and a warning is returned. If an indicator variable is not provided, an error is returned and no more values are assigned to variables. It is possible that some values have already been assigned to variables and will remain assigned when the error is returned.

When a datetime value is returned, the length of the variable must be large enough to store the complete value. Otherwise, depending on how much of the

118. If assigning to an SQL-variable or SQL-parameter and the standards option is specified, storage assignment rules apply. For information about the standards option, see "Standards compliance" on page xi.

value would have to be truncated, a warning or an error is returned. See "Datetime assignments" on page 105 for details.

Examples

Example 1: Using a COBOL program statement, put the maximum salary (SALARY) from the EMPLOYEE table into the host variable MAX-SALARY (DECIMAL(9,2)).

```
EXEC SQL  SELECT MAX(SALARY)
            INTO :MAX-SALARY
            FROM EMPLOYEE WITH CS
END-EXEC.
```

Example 2: Using a Java program statement, select the row from the EMPLOYEE table on the connection context 'ctx' with a employee number (EMPNO) value the same as that stored in the host variable HOST_EMP (java.lang.String). Then put the last name (LASTNAME) and education level (EDLEVEL) from that row into the host variables HOST_NAME (String) and HOST_EDUCATE (Integer).

```
#sql [ctx] {  SELECT LASTNAME, EDLEVEL
              INTO :HOST_NAME, :HOST_EDUCATE
              FROM EMPLOYEE
              WHERE EMPNO = :HOST_EMP   };
```

Example 3: Put the row for employee 528671, from the EMPLOYEE table, into the host structure EMPREC. Assume that the row will be updated later and should be locked when the query executes.

```
EXEC SQL  SELECT *
            INTO :EMPREC
            FROM EMPLOYEE
            WHERE EMPNO = '528671'
            WITH RS USE AND KEEP EXCLUSIVE LOCKS
END-EXEC.
```

SET CONNECTION

The SET CONNECTION statement establishes the current server of the activation group by identifying one of its existing connections.

Invocation

This statement can only be embedded within an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

SET CONNECTION is not allowed in a trigger or function.

Authorization

If a global variable is referenced in a statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax

Description

server-name **or** *variable*
Identifies the connection by the specified server name or the server name contained in the variable. It can be a global variable if it is qualified with schema name. If a variable is specified:

- It must be a character-string variable with a length attribute that is not greater than 18.
- It must not be followed by an indicator variable.
- The server name must be left-justified within the variable and must conform to the rules for forming an ordinary identifier.
- If the length of the server name is less than the length of the variable, it must be padded on the right with blanks.

Let S denote the specified server name or the server name contained in the variable. S must identify an existing connection of the application process. If S identifies the current connection, the state of S and all other connections of the application process are unchanged, but information about S is placed in the SQLERRP field of the SQLCA. The following rules apply when S identifies a dormant connection.

If the SET CONNECTION statement is successful:

- Connection S is placed in the current state.
- S is placed in the CURRENT SERVER special register.
- Information about the application server is placed in the *connection-information-items* in the SQL Diagnostics Area.

- Information about application server S is also placed in the SQLERRP field of the SQLCA. If the application server is an IBM relational database product, the information has the form *pppvvrrm*, where:
  - *ppp* identifies the product as follows:
    - ARI for DB2 for VSE and VM
    - DSN for DB2 for z/OS
    - QSQ for DB2 for i
    - SQL for all other DB2 products
  - *vv* is a two-digit version identifier such as '04'
  - *rr* is a two-digit release identifier such as '01'
  - *m* is a one-digit modification level such as '0'

  For example, if the application server is Version 4 of DB2 for z/OS, the value of SQLERRP is 'DSN04010'.
- Additional information about the connection is available from the DB2_CONNECTION_STATUS and DB2_CONNECTION_TYPE connection information items in the SQL Diagnostics Area.

  The DB2_CONNECTION_STATUS connection information item indicates the status of connection for this unit of work. It will have one of the following values:
  - 1 - Committable updates can be performed on the connection for this unit of work.
  - 2 - No committable updates can be performed on the connection for this unit of work.

  The DB2_CONNECTION_TYPE connection information item indicates the type of connection. It will have one of the following values:
  - 1 - Connection is to a local relational database.
  - 2 - Connection is to a remote relational database with the conversation unprotected.
  - 3 - Connection is to a remote relational database with the conversation protected.
  - 4 - Connection is to an application requester driver program.
- Additional information about the connection is also placed in the SQLERRD(4) field of the SQLCA. SQLERRD(4) will contain a value indicating whether the application server allows committable updates to be performed. Following is a list of values and their meanings for the SQLERRD(4) field of the SQLCA on the CONNECT :
  - 1 - Committable updates can be performed and either the connection uses an unprotected conversation, is a connection established to an application requester driver program using a CONNECT (Type 1) statement, or is a local connection established using a CONNECT (Type 1) statement.
  - 2 - No committable updates can be performed; conversation is unprotected.
  - 3 - It is unknown if committable updates can be performed; conversation is protected.
  - 4 - It is unknown if committable updates can be performed; conversation is unprotected.
  - 5 - It is unknown if committable updates can be performed and the connection is either a local connection established using a CONNECT (Type 2) statement or a connection to an application requester driver program established using a CONNECT (Type 2) statement.

- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. Refer to Appendix B, "SQL Communication Area" for a description of the information in the SQLERRMC field.
- Any previously current connection is placed in the dormant state.

If the SET CONNECTION statement is unsuccessful, the connection state of the activation group and the states of its connections are unchanged.

Notes

SET CONNECTION for CONNECT (Type 1): The use of CONNECT (Type 1) statements does not prevent the use of SET CONNECTION, but the statement either fails or does nothing because dormant connections do not exist.

Status after connection is restored: When a connection is used, made dormant, and then restored to the current state in the same unit of work, the status of locks, cursors, and prepared statements for that connection reflects its last use by the activation group.

Local connections: A SET CONNECTION to a local connection will fail if the current independent auxiliary Storage pool (IASP) name space does not match the local connection's relational database.

Example

Execute SQL statements at TOROLAB1, execute SQL statements at TOROLAB2, and then execute more SQL statements at TOROLAB1.

```
EXEC SQL CONNECT TO TOROLAB1;

(Execute statements referencing objects at TOROLAB1)

EXEC SQL CONNECT TO TOROLAB2;

(Execute statements referencing objects at TOROLAB2)

EXEC SQL SET CONNECTION TOROLAB1;

(Execute statements referencing objects at TOROLAB1)
```

The first CONNECT statement creates the TOROLAB1 connection, the second CONNECT statement places it in the dormant state, and the SET CONNECTION statement returns it to the current state.

SET CURRENT DEBUG MODE

The SET CURRENT DEBUG MODE statement assigns a value to the CURRENT DEBUG MODE special register.

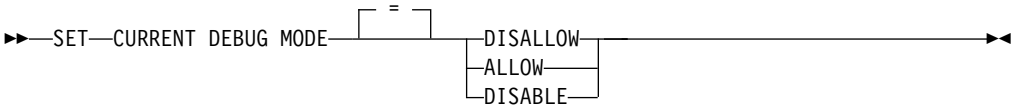
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None.

Syntax



Description

The value of CURRENT DEBUG MODE is replaced by the specified keyword:

DISALLOW
Procedures will be created so they cannot be debugged by the Unified Debugger. When the DEBUG MODE attribute of a procedure is DISALLOW, the procedure can be subsequently altered to change the DEBUG MODE attribute.

ALLOW
Procedures will be created so they can be debugged by the Unified Debugger. When the DEBUG MODE attribute of a procedure is ALLOW, the procedure can be subsequently altered to change the DEBUG MODE attribute.

DISABLE
Procedures will be created so they cannot be debugged by the Unified Debugger. When the DEBUG MODE attribute of a procedure is DISABLE, the procedure cannot be subsequently altered to change the DEBUG MODE attribute.

Notes

Transaction considerations: The SET CURRENT DEBUG MODE statement is not a committable operation. ROLLBACK has no effect on the current debug mode.

Initial current debug mode: The initial value of the current debug mode is DISALLOW.

Current debug mode scope: The scope of the current debug mode is the job.

Example

Example 1: The following statement sets the CURRENT DEBUG MODE to allow subsequent procedures created by the CREATE PROCEDURE (SQL) statement to be debuggable.

```
SET CURRENT DEBUG MODE = ALLOW
```

Example 2: The following statement sets the CURRENT DEBUG MODE to disallow subsequent procedures created by the CREATE PROCEDURE (SQL) statement to be debuggable and to prevent those procedures from being altered to make them debuggable.

```
SET CURRENT DEBUG MODE = DISABLE
```

SET CURRENT DECFLOAT ROUNDING MODE

The SET CURRENT DECFLOAT ROUNDING MODE statement changes the value of the CURRENT DECFLOAT ROUNDING MODE special register.

Invocation

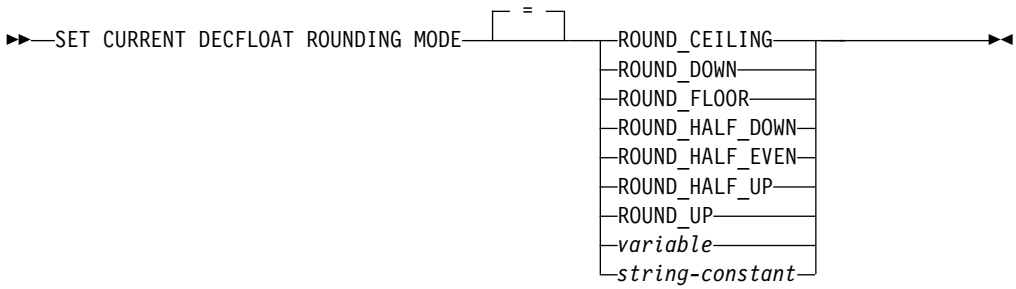
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

If a global variable is referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Description

ROUND_CEILING
Round toward +Infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by one (rounded up).

ROUND_DOWN
Round toward zero (truncation). The discarded digits are ignored.

ROUND_FLOOR
Round toward -Infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged other than the removal of the discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by one.

ROUND_HALF_DOWN
Round to nearest; if equidistant, round down. If the discarded digits represent greater than half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). Otherwise, the discarded digits are ignored.

This rounding mode is not recommended when creating a portable application since it is not supported by the IEEE draft standard for floating-point arithmetic.

ROUND_HALF_EVEN
Round to nearest; if equidistant, round so that the final digit is even. If the discarded digits represent greater than half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise (they represent exactly half), the result coefficient is unaltered if its rightmost digit is even or incremented by one (rounded up) if its rightmost digit is odd (to make an even digit).

ROUND_HALF_UP
Round to nearest; if equidistant, round up. If the discarded digits represent greater than or equal to half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). Otherwise, the discarded digits are ignored.

ROUND_UP
Round away from zero. If all of the discarded digits are zero, the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by one (rounded up).

This rounding mode is not recommended when creating a portable application since it is not supported by the IEEE draft standard for floating-point arithmetic.

string-constant
A character constant that contains a specification of the rounding mode.

variable
Specifies a variable which contains the value for the CURRENT DECFLOAT ROUNDING MODE. The content is not folded to uppercase.

The variable:

- Must be a character-string or Unicode graphic-string variable.
- Must not be followed by an indicator variable.
- Must contain one of the seven rounding mode keywords.
- Must be padded on the right with blanks if the variable is fixed length.

Notes

Transaction considerations: The SET CURRENT DECFLOAT ROUNDING MODE statement is not a committable operation. ROLLBACK has no effect on the CURRENT DECFLOAT ROUNDING MODE.

Initial CURRENT DECFLOAT ROUNDING MODE: The initial value of CURRENT DECFLOAT ROUNDING MODE in an activation group is established by the first SQL statement that is executed in the activation group.

- If the first SQL statement in an activation group is executed from an SQL program or SQL package, the CURRENT DECFLOAT ROUNDING MODE special register is set to the value of the DECFLTRND parameter.
- Otherwise, the initial value is ROUND_HALF_EVEN.

CURRENT DECFLOAT ROUNDING MODE scope is activation group.

Examples

Example 1: Set the CURRENT DECFLOAT ROUNDING MODE special register to ROUND_DOWN using a string constant and using a keyword.

```
SET CURRENT DECFLOAT ROUNDING MODE = 'ROUND_DOWN'
SET CURRENT DECFLOAT ROUNDING MODE = ROUND_DOWN
```

SET CURRENT DEGREE

The SET CURRENT DEGREE statement assigns a value to the CURRENT DEGREE special register.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared. It must not be specified in REXX.

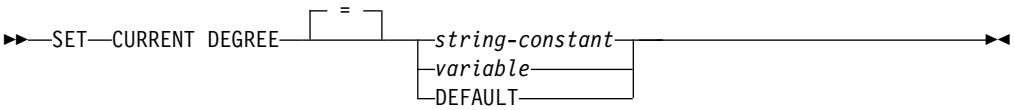
Authorization

The privileges held by the authorization ID of the statement must include *JOBCTL special authority or be authorized to the SQL Administrator function of IBM i through Application Administration in System i Navigator. The Change Function Usage Information (CHGFCNUSG) command, with a function ID of QIBM_DB_SQLADM, can also be used to change the list of authorized users.

If a global variable is referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Description

The value of CURRENT DEGREE is replaced by the value of the string constant or variable.

string-constant
Specifies a character string constant. The content is not folded to uppercase.

The length of the *string-constant* must not exceed 5 after trimming any leading and trailing blanks.

variable
Specifies a variable that contains the value for CURRENT DEGREE.

The variable:

- Must be a CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable. The actual length of the contents of the *variable* must not be greater than 5 after trimming any leading and trailing blanks.
- Must not be the null value.
- Must have contents in uppercase characters. All characters are case-sensitive and are not converted to uppercase characters.

DEFAULT
If the PARALLEL_DEGREE parameter in a current query options file (QAQQINI) is specified, the CURRENT DEGREE will be reset to the PARALLEL_DEGREE. Otherwise, the CURRENT DEGREE will be reset from the degree specified by the QQRYDEGREE system value.

The value of the string constant or variable must be one of the following:

1 No parallel processing is allowed.

2 through 32767
Specifies the degree of parallelism that will be used.

ANY
Specifies that the database manager can choose to use any number of tasks for either I/O or SMP parallel processing.

Use of parallel processing and the number of tasks used is determined based on the number of processors available in the system, this job's share of the amount of active memory available in the pool in which the job is run, and whether the expected elapsed time for the operation is limited by CPU processing or I/O resources. The database manager chooses an implementation that minimizes elapsed time based on the job's share of the memory in the pool.

NONE
No parallel processing is allowed.

MAX
The database manager can choose to use any number of tasks for either I/O or SMP parallel processing. MAX is similar to ANY except the database manager assumes that all active memory in the pool can be used.

IO Any number of tasks can be used when the database manager chooses to use I/O parallel processing for queries. SMP is not allowed.

Notes

Transaction considerations: The SET CURRENT DEGREE statement is not a committable operation. ROLLBACK has no effect on CURRENT DEGREE.

Initial current degree: The initial value of CURRENT DEGREE is equal to the parallelism degree in effect from the CHGQRYA CL command, PARALLEL_DEGREE parameter in the current query options file (QAQQINI), or the QQRYDEGREE system value.

Parallelism degree precedence: The parallelism degree can be controlled in several ways. The actual parallelism degree used is determined as follows:

- If a SET CURRENT DEGREE statement or a CHGQRYA CL command with a DEGREE keyword has been executed, the parallelism degree specified by the most recent of either is the value of CURRENT DEGREE.
- If neither a SET CURRENT DEGREE statement nor a CHGQRYA CL command with a DEGREE keyword has been executed,
  - If a current query options file (QAQQINI) with a PARALLEL_DEGREE parameter has been specified, the parallelism degree specified by the QAQQINI file is the value of CURRENT DEGREE.
  - Otherwise, the parallelism degree specified by the QQRYDEGREE system value is the value of CURRENT DEGREE.

For more information, see Database Performance and Query Optimization topic collection.

Current degree scope: The scope of CURRENT DEGREE is the job.

Parallel limitations: If the DB2 Symmetric Multiprocessing feature is not installed, a warning is returned and parallelism is not used.

Some SQL statements cannot use parallelism.

Example

Example 1: The following statement sets the CURRENT DEGREE to inhibit parallelism.

```
SET CURRENT DEGREE = '1'
```

Example 2: The following statement sets the CURRENT DEGREE to allow parallelism.

```
SET CURRENT DEGREE = 'ANY'
```

SET CURRENT IMPLICIT XMLPARSE OPTION

The SET CURRENT IMPLICIT XMLPARSE OPTION statement changes the value of the CURRENT IMPLICIT XMLPARSE OPTION special register.

Invocation

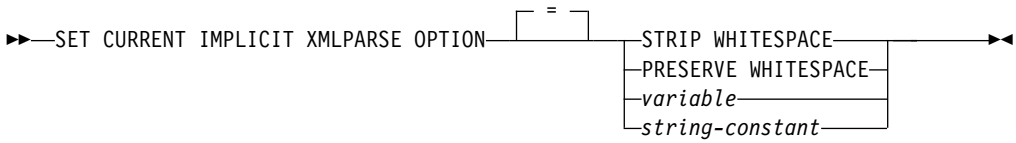
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

If a global variable is referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Description

STRIP WHITESPACE
Whitespace is removed on implicit XMLPARSE.

PRESERVE WHITESPACE
Whitespace is not removed on implicit XMLPARSE.

variable
Specifies a variable which contains the value for the CURRENT IMPLICIT XMLPARSE OPTION. The content is folded to uppercase.

The variable:

- Must be a character-string or Unicode graphic-string variable.
- Must not be followed by an indicator variable.
- Must contain one of the two implicit XMLPARSE options.
- Must be padded on the right with blanks if the variable is fixed length.

string-constant
A character constant that contains a specification of the implicit XMLPARSE option. The value must be a left justified string that is either 'STRIP WHITESPACE' or 'PRESERVE WHITESPACE' with exactly one blank character between the keywords. The content is folded to uppercase.

Notes

Transaction considerations: The SET CURRENT IMPLICIT XMLPARSE OPTION statement is not a committable operation. ROLLBACK has no effect on the CURRENT IMPLICIT XMLPARSE OPTION.

Initial CURRENT IMPLICIT XMLPARSE OPTION: The initial value of CURRENT IMPLICIT XMLPARSE OPTION is 'STRIP WHITESPACE'.

Both static and dynamic statements are affected by this special register.

The CURRENT IMPLICIT XMLPARSE OPTION scope is the connection.

Example

Set the value of the CURRENT IMPLICIT XMLPARSE OPTION special register to 'PRESERVE WHITESPACE'.

```
SET CURRENT IMPLICIT XMLPARSE OPTION = PRESERVE WHITESPACE
```

SET DESCRIPTOR

The SET DESCRIPTOR statement sets information in an SQL descriptor.

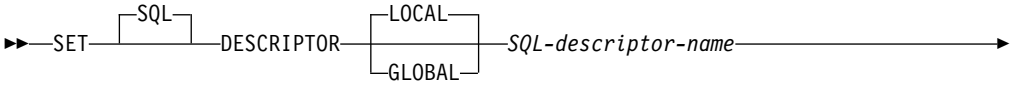
Invocation

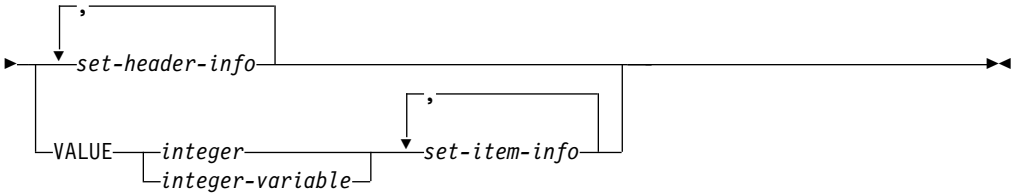
This statement can only be embedded in an application program, SQL function, SQL procedure, or trigger. It cannot be issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

Authorization

None required.

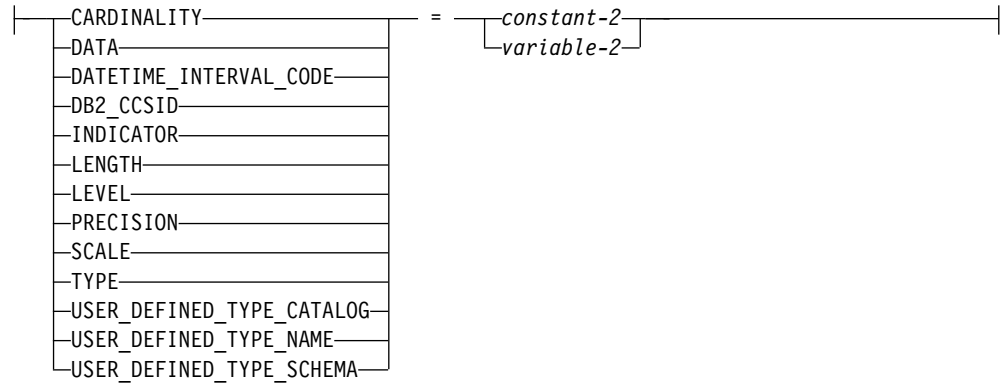
Syntax





set-header-info:

set-item-info:



Description

LOCAL
Specifies the scope of the name of the descriptor to be local to program invocation. The information provided is set into the descriptor known in this local scope.

GLOBAL
Specifies the scope of the name of the descriptor to be global to the SQL session. The information provided is set into the descriptor known to any program that executes using the same database connection.

SQL-descriptor-name
Names the SQL descriptor. The name must identify a descriptor that already exists with the specified scope.

set-header-info
Sets attributes into the SQL descriptor. The same descriptor item must not be specified more than once in a single SET DESCRIPTOR statement.

VALUE
Specifies the item number for which the specified information is set. If the item number is greater than the maximum number of items allocated for the descriptor or the item number is less than 1, an error is returned.

integer
An integer constant in the range of 1 to the number of items allocated in the SQL descriptor.

integer-variable
Identifies a variable declared in the program in accordance with the rules for declaring variables. It must not be a global variable. The data type of the variable must be SMALLINT, INTEGER, BIGINT, or DECIMAL or NUMERIC with a scale of zero. The value of variable must be in the range of 1 to the maximum number of items allocated in the SQL descriptor.

set-item-info
Sets information about a specific item into the SQL descriptor. The same descriptor item must not be specified more than once in a single SET DESCRIPTOR statement. Items that are not applicable to the specified type are ignored.

set-header-info

COUNT
A count of the number of items that will be specified in the descriptor.

variable–1
Identifies a variable declared in the program in accordance with the rules for declaring variables, but must not be a file reference variable or a global variable. The data type of the variable must be compatible with the COUNT header item as specified in Table 83 on page 1190. The variable is assigned (using storage assignment rules) to the COUNT header item. For details on the assignment rules, see “Assignments and comparisons” on page 98.

constant–1
Identifies a constant value used to set the COUNT header item. The data type of the constant must be compatible with the COUNT header item as specified in Table 83 on page 1190. The constant is assigned (using storage assignment rules) to the COUNT header item. For details on the assignment rules, see “Assignments and comparisons” on page 98.

set-item-info

CARDINALITY
Specifies the cardinality for the item. This is only allowed when TYPE is an array.

DATA
Specifies the value for the data described by the item descriptor. If the value of INDICATOR is negative, then the value of DATA is undefined. The assigned value cannot be a constant.

DATETIME_INTERVAL_CODE
Specifies the specific datetime data type. DATETIME_INTERVAL_CODE must be specified if TYPE is set to 9.
1 DATE
2 TIME
3 TIMESTAMP

DB2_CCSID
Specifies the CCSID of character, graphic, XML, or datetime data. The value is not applicable for all other data types. If the DB2_CCSID is not specified or 0 is specified:
- For XML data, the SQL_XML_DATA_CCSID QAQQINI option setting will be used.
- Otherwise, the CCSID of the variable will be determined by the CCSID of the job.

INDICATOR
Specifies the value for the indicator. A non-negative indicates a DATA value will be provided for this descriptor item. When extended indicator variables are not enabled, a negative value indicates the value described by this descriptor item is the null value. If not set, the value of INDICATOR is 0. When extended indicator variables are enabled:
- -1, -2, -3, -4, or -6 indicates the value described by this descriptor item is the null value.
- -5 indicates the value described by this descriptor item is the DEFAULT value.
- -7 indicates the value described by this descriptor item is the UNASSIGNED value.
- 0 or a positive value indicates a DATA value will be provided for this descriptor item.

LENGTH
Specifies the maximum length of the data. If the data type is a character or graphic string type, XML type, or a datetime type, the length represents the number of characters (not bytes). If the data type is a binary string or any other type, the length represents the number of bytes. If LENGTH is not specified, a default length will be used. For a description of the defaults, see Table 109 on page 1365.

LEVEL
The level of the item descriptor.
0 Item is a primary descriptor entry.
1 Item is for a secondary descriptor entry. This is for an array entry.

PRECISION
Specifies the precision for descriptor items of data type DECIMAL, NUMERIC, DECFLOAT, DOUBLE, REAL, and FLOAT. If PRECISION is not specified, a default precision will be used. For a description of the defaults, see Table 109 on page 1365.

SCALE
Specifies the scale for descriptor items of data type DECIMAL or NUMERIC. If SCALE is not specified, a default scale will be used. For a description of the defaults, see Table 109 on page 1365.

TYPE
Specifies a data type code representing the data type of the descriptor item. For a description of the data type codes and lengths, see Table 84 on page 1192. Either TYPE or USER_DEFINED_TYPE_NAME and USER_DEFINED_TYPE_SCHEMA (but not both) must be specified for each descriptor item.

USER_DEFINED_TYPE_CATALOG
Specifies the server name of the user-defined type. If USER_DEFINED_TYPE_CATALOG is specified, it must be equal to the current server. Otherwise, the USER_DEFINED_TYPE_CATALOG is the current server.

USER_DEFINED_TYPE_NAME
Specifies the name of the user-defined data type. Either TYPE or USER_DEFINED_TYPE_NAME and USER_DEFINED_TYPE_SCHEMA (but not both) must be specified for each descriptor item.

USER_DEFINED_TYPE_SCHEMA
Specifies the schema containing the user-defined type. Either TYPE or USER_DEFINED_TYPE_NAME and USER_DEFINED_TYPE_SCHEMA (but not both) must be specified for each descriptor item.

variable-2
Identifies a variable declared in the program in accordance with the rules for declaring variables, but must not be a file reference variable or a global variable. The data type of the variable must be compatible with the descriptor information item as specified in Table 83 on page 1190. The variable is assigned (using storage assignment rules) to the corresponding descriptor item. For details on the assignment rules, see "Assignments and comparisons" on page 98.

When setting the DATA item, in general the variable must have the same data type, length, precision, scale, and CCSID as specified in Table 83 on page 1190. For variable-length types, the variable length must not be less than the LENGTH in the descriptor. For C nul-terminated types, the variable length must be at least one greater than the LENGTH in the descriptor.

constant-2
Identifies a constant value used to set the descriptor item. The data type of the constant must have the same data type, length, precision, scale, and CCSID as specified in Table 83 on page 1190. The constant is assigned (using storage assignment rules) to the corresponding descriptor item. For details on the assignment rules, see "Assignments and comparisons" on page 98.

If the descriptor item to be set is DATA, *constant-2* cannot be specified.

Notes

Default values for descriptor items: The following table represents the default values for LENGTH, PRECISION, and SCALE, if they are not specified for a descriptor item.

Table 109. Default LENGTH, PRECISION, and SCALE

Data Type	LENGTH	PRECISION	SCALE
DECIMAL and NUMERIC		5	0
FLOAT		53	0
DECFLOAT		34	
CHARACTER, VARCHAR, and CLOB	1		
GRAPHIC, VARGRAPHIC, and DBCLOB	1		
BINARY, VARBINARY, and BLOB	1		
DATE	10		
TIME	8		
TIMESTAMP	26		
XML	1		

Example

Example 1: Set the number of items in descriptor 'NEWDA' to the value in :numitems.

```
EXEC SQL SET DESCRIPTOR  'NEWDA'
  COUNT = :numitems;
```

Example 2: Set the value of the type and length for the first item descriptor of descriptor 'NEWDA'

```
SET DESCRIPTOR 'NEWDA'
  VALUE 1 TYPE   = :dtype,
          LENGTH = :olength;
```

SET ENCRYPTION PASSWORD

The SET ENCRYPTION PASSWORD statement sets the default password and hint that will be used by the encryption and decryption functions. The password is not associated with authentication and is only used for data encryption and decryption.

For information about using this statement, see "ENCRYPT_AES" on page 346, "ENCRYPT_RC2" on page 349, "ENCRYPT_TDES" on page 352, and "DECRYPT_BIT, DECRYPT_BINARY, DECRYPT_CHAR and DECRYPT_DB" on page 329.

Invocation

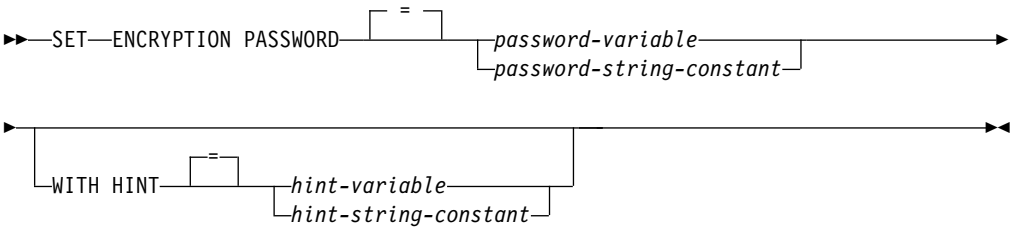
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

If a global variable is referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Description

password-variable
Specifies a variable that contains an encryption password.

The variable:

- Must be a CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable. The actual length of the contents of the variable must be between 6 and 127 inclusive or must be an empty string. If an empty string is specified, the default encryption password is set to no value.
- Must not be the null value.
- All characters are case-sensitive and are not converted to uppercase characters.

password-string-constant
A character constant. The length of the constant must be between 6 and 127 inclusive or must be an empty string. If an empty string is specified, the

default encryption password is set to no value. The literal form of the password is not allowed in static SQL or REXX.

WITH HINT
Indicates that a value is specified that will help data owners remember passwords (for example, 'Ocean' as a hint to remember 'Pacific'). If a hint value is specified, the hint is used as the default for encryption functions. The hint can subsequently be retrieved for an encrypted value using the GETHINT function. If this clause is not specified and a hint is not explicitly specified on the encryption function, no hint will be embedded in encrypted data result.

hint-variable
Specifies a variable that contains an encryption password hint.

The variable:

- Must be a CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable. The actual length of the contents of the variable must not be greater than 32. If an empty string is specified, the default encryption password hint is set to no value.
- Must not be the null value.
- All characters are case-sensitive and are not converted to uppercase characters.

hint-string-constant
A character constant. The length of the constant must not be greater than 32. If an empty string is specified, the default encryption password hint is set to no value.

Notes

Password protection: To prevent inadvertent access to the encryption password, do not specify *password-string-constant* in the source for a program, procedure, or function. Instead, use a variable.

When connected to a remote relational database, the specified password itself is sent "in the clear". That is, the password itself is not encrypted. To protect the password in these cases, consider using a communications encryption mechanism such as IPSEC (or SSL if connecting between IBM i products).

Transaction considerations: The SET ENCRYPTION PASSWORD statement is not a committable operation. ROLLBACK has no effect on the default encryption password or default encryption password hint.

Initial encryption password value: The initial value of both the default encryption password and the default encryption password hint is the empty string ('').

Encryption password scope: The scope of the default encryption password and default encryption password hint is the activation group and connection.

Example

Set the ENCRYPTION PASSWORD to the value in :hv1.

```
SET ENCRYPTION PASSWORD :hv1
```

SET OPTION

The SET OPTION statement establishes the processing options to be used for SQL statements.

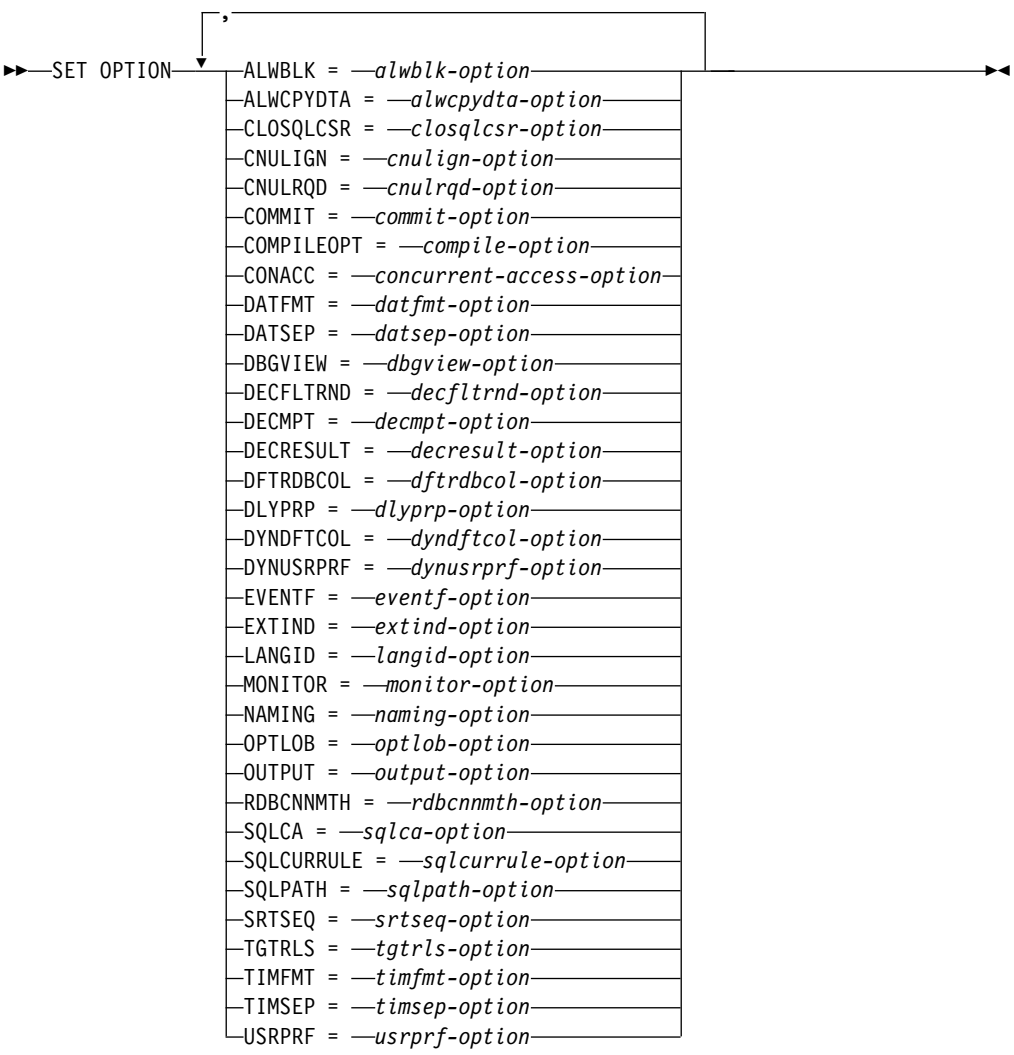
Invocation

This statement can be used in a REXX procedure or embedded in an application program. If used in a REXX procedure, it is an executable statement. If embedded in an application program, it is not executable and must precede any other SQL statements. This statement cannot be dynamically prepared.

Authorization

None required.

Syntax

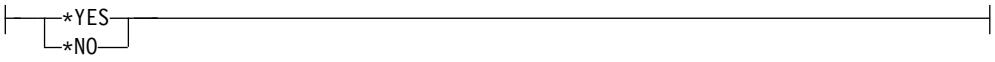


alwblk-option:

alwcpydta-option:

closqlcsr-option:

cnulign-option:

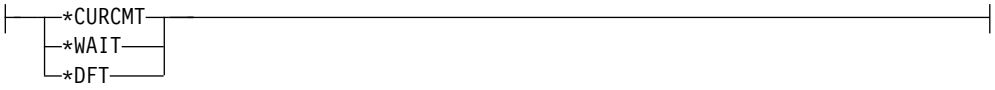


cnulrqd-option:

commit-option:

compile-option:

concurrent-access-option:



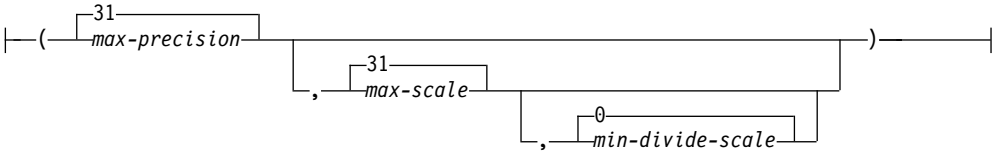
datfmt-option:

datsep-option:

decfltrnd-option:

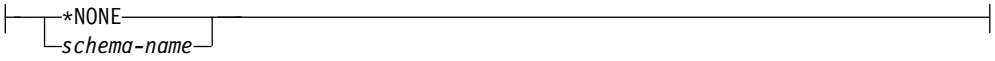
decmpt-option:

decresult-option:



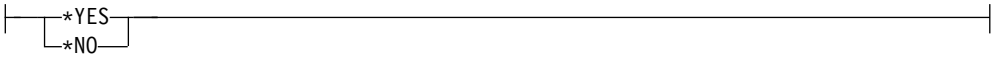
dbgview-option:

dftrdbcol-option:



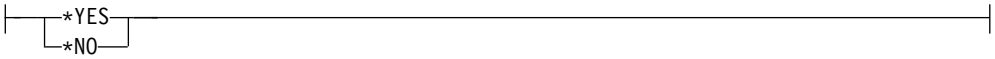
dlyprp-option:

dyndftcol-option:



dynusrprf-option:

eventf-option:

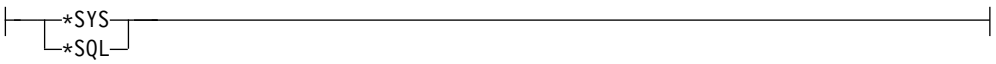


extind-option:

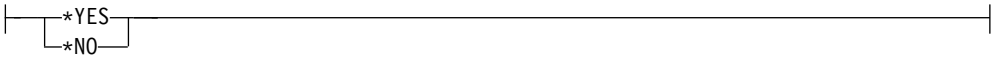
langid-option:

monitor-option:

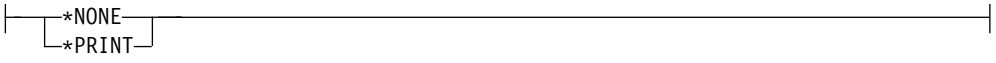
naming-option:



optlob-option:



output-option:



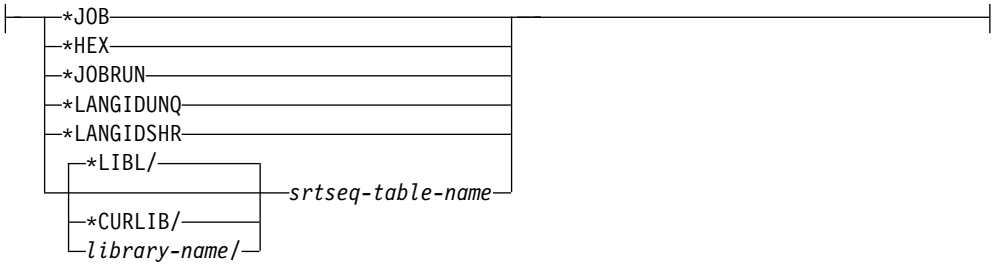
rdbcnnmth-option:

sqlca-option:

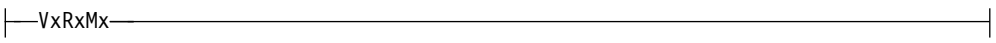
sqlcurrule-option:

sqlpath-option:

srtseq-option:



tgtrls-option:



timfmt-option:

timsep-option:

usrprf-option:

Description

ALWBLK

Specifies whether the database manager can use row blocking and the extent to which blocking can be used for read-only cursors. This option will be ignored in REXX.

***ALLREAD**

Rows are blocked for read-only cursors if COMMIT is *NONE, *CHG, or *CS. All cursors in a program that are not explicitly able to be updated are opened for read-only processing even though EXECUTE or EXECUTE IMMEDIATE statements may be in the program.

Specifying *ALLREAD:

- Allows row blocking under commitment control level *CHG and *CS in addition to the blocking allowed for *READ.
- Can improve the performance of almost all read-only cursors in programs, but limits queries in the following ways:
  - The Rollback (ROLLBACK) command, a ROLLBACK statement in host languages, or the ROLLBACK HOLD SQL statement does not reposition a read-only cursor when:
    - ALWBLK(*ALLREAD) was specified when the program or routine that contains the cursor was created
    - ALWBLK(*READ) and ALWCPYDTA(*OPTIMIZE) were specified when the program or routine that contains the cursor was created

– Dynamic running of a positioned UPDATE or DELETE statement (for example, using EXECUTE IMMEDIATE), cannot be used to update a row in a cursor unless the DECLARE statement for the cursor includes the FOR UPDATE clause.

***NONE**
Rows are not blocked for retrieval of data for cursors.

Specifying *NONE:

- Guarantees that the data retrieved is current.
- May reduce the amount of time required to retrieve the first row of data for a query.
- Stops the database manager from retrieving a block of data rows that is not used by the program when only the first few rows of a query are retrieved before the query is closed.
- Can degrade the overall performance of a query that retrieves a large number of rows.

***READ**
Rows are blocked for read-only retrieval of data for cursors when:

- *NONE is specified on the COMMIT parameter, which indicates that commitment control is not used.
- The cursor is declared with a FOR READ ONLY clause or there are no dynamic statements that could run a positioned UPDATE or DELETE statement for the cursor.

Specifying *READ can improve the overall performance of queries that meet the above conditions and retrieve a large number of rows.

ALWCPYDTA
Specifies whether a copy of the data can be used in a SELECT statement. This option will be ignored in REXX.

***OPTIMIZE**
The system determines whether to use the data retrieved directly from the database or to use a copy of the data. The decision is based on which method provides the best performance. If COMMIT is *CHG or *CS and ALWBLK in not *ALLREAD, or if COMMIT is *ALL or *RR, then a copy of the data is used only when it is necessary to run a query.

***YES**
A copy of the data is used only when necessary.

***NO**
A copy of the data is not allowed. If a temporary copy of the data is required to perform the query, an error message is returned.

CLOSQLCSR
Specifies when SQL cursors are implicitly closed, SQL prepared statements are implicitly discarded, and LOCK TABLE locks are released. SQL cursors are explicitly closed when you issue the CLOSE, COMMIT, or ROLLBACK (without HOLD) SQL statements. This option will be ignored in REXX. *ENDACTGRP and *ENDMOD are for use by ILE programs and modules, SQL functions, SQL procedures, or SQL triggers. *ENDPGM, *ENDSQL, and *ENDJOB are for use by non-ILE programs.

SQL scalar functions, SQL procedures, and SQL triggers use *ENDMOD as the default. SQL table functions are always created using *ENDACTGRP.

***ENDACTGRP**
SQL cursors are closed, SQL prepared statements are implicitly discarded, and LOCK TABLE locks are released when the activation group ends.

***ENDMOD**
SQL cursors are closed and SQL prepared statements are implicitly discarded when the module is exited. LOCK TABLE locks are released when the first SQL program on the call stack ends. Note that a cursor may only be logically closed and will only be physically closed when the first program with SQL leaves the stack and only if that program was not compiled with *ENDACTGRP.

***ENDPGM**
SQL cursors are closed and SQL prepared statements are discarded when the program ends. LOCK TABLE locks are released when the first SQL program on the call stack ends.

***ENDSQL**
SQL cursors remain open between calls and can be fetched without running another SQL OPEN. One of the programs higher on the call stack must have run at least one SQL statement. SQL cursors are closed, SQL prepared statements are discarded, and LOCK TABLE locks are released when the first SQL program on the call stack ends. If *ENDSQL is specified for a program that is the first SQL program called (the first SQL program on the call stack), the program is treated as if *ENDPGM was specified.

***ENDJOB**
SQL cursors remain open between calls and can be fetched without running another SQL OPEN. The programs higher on the call stack do not need to have run SQL statements. SQL cursors are left open, SQL prepared statements are preserved, and LOCK TABLE locks are held when the first SQL program on the call stack ends. SQL cursors are closed, SQL prepared statements are discarded, and LOCK TABLE locks are released when the job ends.

CNULIGN
Specifies whether a NUL-terminator is ignored for character and graphic host variables. This option will only be used for SQL statements in C and C++ programs.

This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

***YES**
Character and graphic host variables defined as NUL-terminated will be treated as fixed length variables for INSERT and UPDATE statements. NUL-terminators are considered part of the data.

***NO**
NUL-terminated character and graphic host variables use NUL-terminators for INSERT and UPDATE statements.

CNULRQD
Specifies whether a NUL-terminator is returned for character and graphic host variables. This option will only be used for SQL statements in C and C++ programs.

This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

***YES**
Output character and graphic host variables always contain the NUL-terminator. If there is not enough space for the NUL-terminator, the

data is truncated and the NUL-terminator is added. Input character and graphic host variables require a NUL-terminator.

***NO**
For output character and graphic host variables, the NUL-terminator is not returned when the host variable is exactly the same length as the data. Input character and graphic host variables do not require a NUL-terminator.

COMMIT
Specifies the isolation level to be used. In REXX, files that are referred to in the source are not affected by this option. Only tables, views, and packages referred to in SQL statements are affected. For more information about isolation levels, see "Isolation level" on page 26

***CHG**
Specifies the isolation level of Uncommitted Read.

***NONE**
Specifies the isolation level of No Commit. If the DROP SCHEMA statement is included in a REXX procedure, *NONE must be used.

***CS**
Specifies the isolation level of Cursor Stability.

***ALL**
Specifies the isolation level of Read Stability.

***RR**
Specifies the isolation level of Repeatable Read.

COMPILEOPT
Specifies additional parameters to be used on the compiler command. The COMPILEOPT string is added to the compiler command built by the precompiler. If 'INCDIR(' is anywhere in the string, the precompiler will call the compiler using the SRCSTMF parameter. The contents of the string is not validated. The compiler command will issue an error if any parameter is incorrect. Using any of the keywords that the precompiler passes to the compiler will cause the compiler command to fail because of duplicate parameters. Refer to the Embedded SQL Programming topic collection for a list of parameters that the precompiler generates for the compiler command. This option will be ignored in REXX.

This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

***NONE**
No additional parameters will be used on the compiler command.

character-string
A character constant of no more than 5000 characters containing the compiler options.

CONACC
Specifies the concurrent access resolution to use for select statements.

***CURCMT**
Specifies that the database manager is to use the currently committed version of data when encountering a row that is in the process of being updated or deleted. Rows that are in the process of being inserted are skipped. This value will be honored when possible for isolation level of *CS.

***WAIT**
Specifies to wait for a commit or rollback for data that is in the process of being updated or deleted by another transaction. This value will be honored when possible for isolation levels of *CS and *ALL.

***DFT**
Specifies that the concurrent access option will not be explicitly set for this program. The value that is in effect when the program is invoked will be used.

DATFMT
Specifies the format used when accessing date result columns. All output date fields are returned in the specified format. For input date strings, the specified value is used to determine whether the date is specified in a valid format.

Note: An input date string that uses the format *USA, *ISO, *EUR, or *JIS is always valid.

***JOB:**
The format specified for the job is used. Use the Display Job (DSPJOB) command to determine the current date format for the job.

***ISO**
The International Organization for Standardization (ISO) date format (yyyy-mm-dd) is used.

***EUR**
The European date format (dd.mm.yyyy) is used.

***USA**
The United States date format (mm/dd/yyyy) is used.

***JIS**
The Japanese Industrial Standard date format (yyyy-mm-dd) is used.

***MDY**
The date format (mm/dd/yy) is used.

***DMY**
The date format (dd/mm/yy) is used.

***YMD**
The date format (yy/mm/dd) is used.

***JUL**
The Julian date format (yy/ddd) is used.

DATSEP
Specifies the separator used when accessing date result columns.

Note: This parameter applies only when *JOB, *MDY, *DMY, *YMD, or *JUL is specified on the DATFMT parameter.

***JOB**
The date separator specified for the job is used. Use the Display Job (DSPJOB) command to determine the current value for the job.

***SLASH or '/'**
A slash (/) is used.

***PERIOD or '.'**
A period (.) is used.

***COMMA or ','**
A comma (,) is used.

***DASH or '-'**
A dash (-) is used.

***BLANK or ' '**
A blank ( ) is used.

DBGVIEW
Specifies whether the object can be debugged by the system debug facilities and the type of debug information to be provided by the compiler. The DBGVIEW parameter can only be specified in the body of SQL functions, procedures, and triggers.

If DEBUG MODE in a CREATE PROCEDURE or ALTER PROCEDURE statement is specified, a DBGVIEW option in the SET OPTION statement must not be specified.

The possible choices are:

***NONE**
A debug view will not be generated.

***SOURCE**
Allows the compiled module object to be debugged using SQL statement source. If *SOURCE is specified, the modified source is stored in source file QSQDSRC in the same schema as the created function, procedure, or trigger.

***STMT**
Allows the compiled module object to be debugged using program statement numbers and symbolic identifiers.

***LIST**
Generates the listing view for debugging the compiled module object.

If DEBUG MODE is not specified, but a DBGVIEW option in the SET OPTION statement is specified, the procedure cannot be debugged by the Unified Debugger, but can be debugged by the system debug facilities. If neither DEBUG MODE nor a DBGVIEW option is specified, the debug mode used is from the CURRENT DEBUG MODE special register.

DECFLTRND
Specifies the DECFLOAT rounding mode used for static SQL statements. The possible choices are:

***CEILING**
Round toward +Infinity. If all of the discarded digits are zero or if the sign is negative the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by one (rounded up).

***DOWN**
Round toward zero (truncation). The discarded digits are ignored.

***FLOOR**
Round toward -Infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged other than the removal of the discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by one.

***HALFDOWN**
Round to nearest; if equidistant, round down. If the discarded digits represent greater than half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). Otherwise, the discarded digits are ignored.

***HALFEVEN**
Round to nearest; if equidistant, round so that the final digit is even. If the discarded digits represent greater than half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise (they represent exactly half), the result coefficient is unaltered if its rightmost digit is even or incremented by one (rounded up) if its rightmost digit is odd (to make an even digit).

***HALFUP**
Round to nearest; if equidistant, round up. If the discarded digits represent greater than or equal to half (0.5) of the value of a one in the next left position, then the result coefficient is incremented by one (rounded up). Otherwise, the discarded digits are ignored.

***UP**
Round away from zero. If all of the discarded digits are zero, the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by one (rounded up).

DECMPT
Specifies the symbol that you want to represent the decimal point. The possible choices are:

***PERIOD**
The representation for the decimal point is a period.

***COMMA**
The representation for the decimal point is a comma.

***SYSVAL**
The representation for the decimal point is the system value (QDECFMT).

***JOB**
The representation for the decimal point is the job value (DECFMT).

DECRESULT
Specifies the maximum precision, maximum scale, and minimum divide scale that should be used during decimal operations, such as decimal arithmetic. The specified limits only apply to NUMERIC and DECIMAL data types.

max-precision
An integer constant that is the maximum precision that should be returned from decimal operations. The value can be 31 or 63. The default is 31.

max-scale
An integer constant that is the maximum scale that should be returned from decimal operations. The value can range from 0 to the maximum precision. The default is 31.

min-divide-scale
An integer constant that is the minimum scale that should be returned from division operations. The value can range from 1 to 9 and cannot be greater than *max-scale*. The default is 0, where 0 indicates that no minimum scale is specified.

DFTRDBCOL
Specifies the schema name used for the unqualified names of tables, views, indexes, and SQL packages. This parameter applies only to static SQL statements. This option will be ignored in REXX.

***NONE**
The naming convention specified on the OPTION precompile parameter or by the SET OPTION NAMING option will be used.

schema-name
Specify the name of the schema. This value is used instead of the naming convention specified on the OPTION precompile parameter or by the SET OPTION NAMING option.

DLYPRP
Specifies whether the dynamic statement validation for a PREPARE statement is delayed until an OPEN, EXECUTE, or DESCRIBE statement is run. Delaying validation improves performance by eliminating redundant validation. This option will be ignored in REXX.

***NO**
Dynamic statement validation is not delayed. When the dynamic statement is prepared, the access plan is validated. When the dynamic statement is used in an OPEN or EXECUTE statement, the access plan is revalidated. Because the authority or the existence of objects referred to by the dynamic statement may change, you must still check the SQLCODE or SQLSTATE after issuing the OPEN or EXECUTE statement to ensure that the dynamic statement is still valid.

***YES**
Dynamic statement validation is delayed until the dynamic statement is used in an OPEN, EXECUTE, or DESCRIBE SQL statement. When the dynamic statement is used, the validation is completed and an access plan is built. If you specify *YES, you should check the SQLCODE and SQLSTATE after running an OPEN, EXECUTE, or DESCRIBE statement to ensure that the dynamic statement is valid.

Note: If you specify *YES, performance is not improved if the INTO clause is used on the PREPARE statement or if a DESCRIBE statement uses the dynamic statement before an OPEN is issued for the statement.

DYNDFTCOL
Specifies the schema name specified for the DFTRDBCOL parameter is also used for dynamic statements. This option will be ignored in REXX.

***NO**
Do not use the value specified for DFTRDBCOL for unqualified names of tables, views, indexes, and SQL packages for dynamic SQL statements. The naming convention specified on the OPTION precompile parameter or by the SET OPTION NAMING option will be used.

***YES**
The schema name specified for DFTRDBCOL will be used for the unqualified names of the tables, views, indexes, and SQL packages in dynamic SQL statements.

DYNUSRPRF
Specifies the user profile to be used for dynamic SQL statements. This option will be ignored in REXX.

***USER**
Local dynamic SQL statements are run under the user profile of the job. Distributed dynamic SQL statements are run under the user profile of the application server job.

***OWNER**
Local dynamic SQL statements are run under the user profile of the program's owner. Distributed dynamic SQL statements are run under the user profile of the SQL package's owner.

EVENTF
Specifies whether an event file will be generated. CoOperative Development Environment/400 (CODE/400) uses the event file to provide error feedback integrated with the CODE/400 editor.

***YES**
The compiler produces an event file for use by CoOperative Development Environment/400 (CODE/400).

***NO**
The compiler will not produce an event file for use by CoOperative Development Environment/400 (CODE/400).

EXTIND
Specifies how to treat indicator variable values passed for SQL statements.

***NO**
Specifies that extended indicator variables are not enabled and non-updatable columns are not allowed in the implicit or explicit UPDATE clause of a *select-statement*

***YES**
Specifies that extended indicator variables are enabled and non-updatable columns are allowed in the implicit or explicit UPDATE clause of a *select-statement*.

LANGID
Specifies the language identifier to be used when SRTSEQ(*LANGIDUNQ) or SRTSEQ(*LANGIDSHR) is specified.

***JOB or *JOBRUN**
The LANGID value for the job is used.

For distributed applications, LANGID(*JOBRUN) is valid only when SRTSEQ(*JOBRUN) is also specified.

language-id
Specify a language identifier to be used. For information about the values that can be used for the language identifier, see the Language identifier topic in the Globalization topic collection.

MONITOR
Specifies whether the statements should be identified as user or system statements when a database monitor is run.

***USER**
The SQL statements are identified as user statements. This is the default.

***SYSTEM**
The SQL statements are identified as system statements.

NAMING
Specifies whether the SQL naming convention or the system naming convention is to be used. This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

The possible choices are:

***SYS**
The system naming convention will be used.

***SQL**
The SQL naming convention will be used.

OPTLOB
Specifies whether accesses to XML and LOBs can be optimized when accessing through DRDA. The possible choices are:

***YES**
LOB and XML accesses should be optimized. The first FETCH for a cursor determines how the cursor will be used for LOBs and XML on all subsequent FETCHes. This option remains in effect until the cursor is closed.

If the first FETCH uses a locator to access a LOB or XML column, no subsequent FETCH for that cursor can fetch that LOB or XML column into a LOB or XML variable.

If the first FETCH places the LOB or XML column into a LOB or XML variable, no subsequent FETCH for that cursor can use a locator for that column.

***NO**
LOB accesses should not be optimized. There is no restriction on whether a column is retrieved into a LOB locator or into a LOB variable. This option can cause performance to degrade.

OUTPUT
Specifies whether the precompiler and compiler listings are generated. The OUTPUT parameter can only be specified in the body of SQL functions, procedures, and triggers. The possible choices are:

***NONE**
The precompiler and compiler listings are not generated.

***PRINT**
The precompiler and compiler listings are generated.

RDBCNNMTH
Specifies the semantics used for CONNECT statements. This option will be ignored in REXX.

***DUW**
CONNECT (Type 2) semantics are used to support distributed unit of work. Consecutive CONNECT statements to additional relational databases do not result in disconnection of previous connections.

***RUW**
CONNECT (Type 1) semantics are used to support remote unit of work. Consecutive CONNECT statements result in the previous connection being disconnected before a new connection is established.

SQLCA
Specifies whether the fields in an SQLCA will be set after each SQL statement.

The SQLCA option is only allowed for ILE C, ILE C++, ILE COBOL, and ILE RPG. This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

The possible choices are:

***YES**
The fields in an SQLCA will be set after each SQL statement. The user program can reference all the values in the SQLCA following the execution of an SQL statement.

***NO**
The fields in an SQLCA will not be set after each SQL statement. The user program should use the GET DIAGNOSTICS statement to retrieve information about the execution of the SQL statement.

SQLCA(*NO) will typically perform better than SQLCA(*YES).

In other host languages, an SQLCA is required and fields in the SQLCA will be set after each SQL statement.

SQLCURRULE
Specifies the semantics used for SQL statements.

***DB2**
The semantics of all SQL statements will default to the rules established for DB2. The following semantics are controlled by this option:

- Hexadecimal constants are treated as character data.
- Unicode graphic-string constants are UCS-2 (CCSID 13488).
- Assignments to SQL-variables and SQL-parameters within the body of a routine or a trigger will use retrieval assignment rules.
- When describing a select statement into an SQLDA and SQLN is smaller than the required number of SQLVAR entries, SQLSTATE 01005 is returned only when the result table contains LOBs or UDTs.

***STD**
The semantics of all SQL statements will default to the rules established by the ISO and ANSI SQL standards. The following semantics are controlled by this option:

- Hexadecimal constants are treated as binary data.
- Unicode graphic-string constants are UTF-16 (CCSID 1200).
- Assignments to SQL-variables and SQL-parameters within the body of a routine or a trigger will use storage assignment rules.
- When describing a select statement into an SQLDA and SQLN is smaller than the required number of SQLVAR entries, SQLSTATE 01005 is always returned.

SQLPATH
Specifies the path to be used to find procedures, functions, and user defined types in static SQL statements. This option will be ignored in REXX.

***LIBL**
The path used is the library list at runtime.

character-string
A character constant with one or more schema names that are separated by commas. Only system schema names can be specified.

SRTSEQ
Specifies the collating sequence table to be used for string comparisons in SQL statements.

Note: *HEX must be specified if a REXX procedure connects to an application server that is not a DB2 for i or a IBM i product whose release level is prior to V2R3M0.

***JOB or *JOBRUN**
The SRTSEQ value for the job is used.

***HEX**
A collating sequence table is not used. The hexadecimal values of the characters are used to determine the collating sequence.

***LANGIDUNQ**
The collating sequence table must contain a unique weight for each character in the code page.

***LANGIDSHR**
The shared-weight sort table for the LANGID specified is used.

srtseq-table-name
Specify the name of the collating sequence table to be used with this program. The name of the collating sequence table can be qualified by one of the following library values:

***LIBL**
All libraries in the user and system portions of the job's library list are searched until the first match is found.

***CURLIB**
The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name
Specify the name of the library to be searched.

TGTRLS
Specifies the release of the operating system on which the user intends to use the object being created. The TGTRLS parameter can only be specified in the body of SQL functions, procedures, and triggers. The possible choices are:

VxRxMx
Specify the release in the format VxRxMx, where Vx is the version, Rx is the release, and Mx is the modification level. For example, V5R4M0 is version 5, release 4, modification level 0. The object can be used on a system with the specified release or with any subsequent release of the operating system installed.

Valid values depend on the current version, release, and modification level, and they change with each new release. If you specify a release-level which is earlier than the earliest release level supported by the database manager, an error message is sent indicating the earliest supported release.

The TGTRLS option can only be specified for SQL functions, SQL procedures, and triggers.

TIMFMT
Specifies the format used when accessing time result columns. All output time fields are returned in the specified format. For input time strings, the specified value is used to determine whether the time is specified in a valid format.

Note: An input time string that uses the format *USA, *ISO, *EUR, or *JIS is always valid.

***HMS**
The (hh:mm:ss) format is used.

***ISO**
The International Organization for Standardization (ISO) time format (hh.mm.ss) is used.

***EUR**
The European time format (hh.mm.ss) is used.

***USA**
The United States time format (hh:mm *xx*) is used, where *xx* is AM or PM.

***JIS**
The Japanese Industrial Standard time format (hh:mm:ss) is used.

TIMSEP
Specifies the separator used when accessing time result columns.

Note: This parameter applies only when *HMS is specified on the TIMFMT parameter.

***JOB**
The time separator specified for the job is used. Use the Display Job (DSPJOB) command to determine the current value for the job.

***COLON or ':'**
A colon (:) is used.

***PERIOD or '.'**
A period (.) is used.

***COMMA or ','**
A comma (,) is used.

***BLANK or ' '**
A blank ( ) is used.

USRPRF
Specifies the user profile that is used when the compiled program object is run, including the authority that the program object has for each object in static SQL statements. The profile of either the program owner or the program user is used to control which objects can be used by the program object. This option will be ignored in REXX.

***NAMING**
The user profile is determined by the naming convention. If the naming convention is *SQL, USRPRF(*OWNER) is used. If the naming convention is *SYS, USRPRF(*USER) is used.

***USER**
The profile of the user running the program object is used.

***OWNER**
The user profiles of both the program owner and the program user are used when the program is run.

Notes

Default values: The default values for the options depend on the language, object type, and the options in effect at create time:

- When an SQL procedure, SQL function, or SQL trigger is created, the default values for the options are those in effect at the time the object is created. For example, if an SQL procedure is created and the current COMMIT option is *CS, *CS is the default COMMIT option. Each option is then updated as it is encountered within the SET OPTION statement.
- For application programs other than REXX, the default values for the options are specified on the CRTSQLxxx command. Each option is then updated as it is encountered within a SET OPTION statement. All SET OPTION statements must precede any other embedded SQL statements.
- At the start of a REXX procedure the options are set to their default value. The default value for each option is the first value listed in the syntax diagram. When an option is changed by a SET OPTION statement, the new value will stay in effect until the option is changed again or the REXX procedure ends.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- *UR can be used as a synonym for *CHG.
- *NC can be used as a synonym for *NONE.
- *RS can be used as a synonym for *ALL.

Examples

Example 1: Set the isolation level to *ALL and the naming mode to SQL names.

```
EXEC SQL SET OPTION COMMIT =*ALL, NAMING =*SQL
```

Example 2: Set the date format to European, the isolation level to *CS, and the decimal point to the comma.

```
EXEC SQL SET OPTION DATFMT = *EUR, COMMIT = *CS, DECMPT = *COMMA
```

SET PATH

The SET PATH statement changes the value of the CURRENT PATH special register.

Invocation

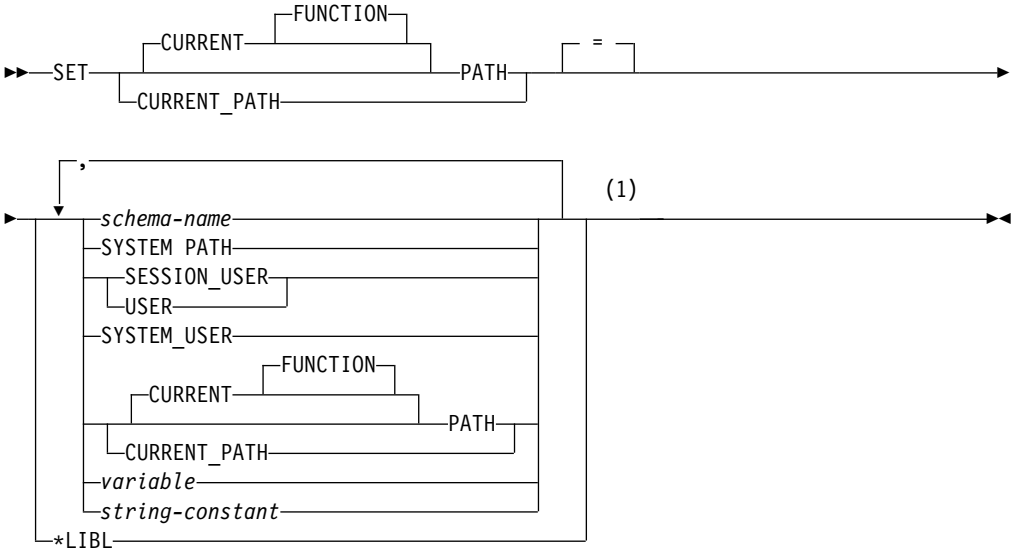
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

If a global variable is referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Notes:

1 SYSTEM PATH, SESSION_USER, USER, SYSTEM_USER, and CURRENT PATH may each be specified at most once on the right side of the statement.

Description

schema-name

Identifies a schema. If a specified schema name is a system schema name, no validation that the schema exists is made at the time the PATH is set. For example, if a *schema-name* is misspelled, it could affect the way subsequent SQL operates. If the specified schema name is not a system schema name, the schema must exist at the time the PATH is set.

Although not recommended, PATH can be specified as a *schema-name* if it is specified as "PATH".

SYSTEM PATH
Specifies the schema names for the system path. This value is the same as specifying the schema names "QSYS","QSYS2","SYSPROC","SYSIBMADM".

SESSION_USER or USER
Specifies the value of the SESSION_USER special register.

SYSTEM_USER
Specifies the value of the SYSTEM_USER special register.

CURRENT PATH
Specifies the value of the CURRENT PATH special register before the execution of this statement. CURRENT PATH is not allowed if the current path is *LIBL.

variable
Specifies a variable that contains one or more schema names that are separated by commas. It can be a global variable if it is qualified with schema name.

The variable:

- Must be a CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable. The actual length of the contents of the *variable* must not exceed the maximum length of a path.
- Must not be followed by an indicator variable.
- Must not be the null value.
- Each schema name must conform to the rules for forming an ordinary or delimited identifier.
- Each schema name must not contain lowercase letters or characters that cannot be specified in an ordinary identifier.
- Must be padded on the right with blanks if the variable is fixed length character.

string-constant
A character constant with one or more schema names that are separated by commas.

The string constant:

- Each schema name must conform to the rules for forming an ordinary or delimited identifier.
- Each schema name must not contain lowercase letters or characters that cannot be specified in an ordinary identifier.

***LIBL**
The path is set to the library list of the current thread.

Notes

Transaction considerations: The SET PATH statement is not a committable operation. ROLLBACK has no effect on the CURRENT PATH.

Rules for the content of the SQL path:

- A schema name must not appear more than once in the path.
- The number of schemas that can be specified is limited by the total length of the CURRENT PATH special register. The special register string is built by taking each schema name specified and removing trailing blanks, delimiting with double quotes, and separating each schema name by a comma. An error is returned if the length of the resulting string exceeds 3483 bytes. A maximum of 268 schema names can be represented in the path.

- There is a difference between specifying a single keyword (such as USER, or PATH, or CURRENT_PATH) as a single keyword, or as a delimited identifier. To indicate that the current value of a special register specified as a single keyword should be used in the SQL path, specify the name of the special register as a keyword. If the name of the special register is specified as a delimited identifier instead (for example, "USER"), it is interpreted as a schema name of that value ('USER'). For example, assuming that the current value of the USER special register is SMITH, then SET PATH = SYSIBM, USER, "USER" results in a CURRENT PATH value of "SYSIBM","SMITH","USER".
- The following rules are used to determine whether a value specified in a SET PATH statement is a variable or a *schema-name*:
  - If *name* is the same as a parameter or SQL variable in the SQL procedure, *name* is interpreted as a parameter or SQL variable, and the value in *name* is assigned to PATH.
  - If *name* is not the same as a parameter or SQL variable in the SQL procedure, *name* is interpreted as *schema-name*, and the value *name* is assigned to PATH.

The system path: SYSTEM PATH refers to the system path for a platform. The schemas QSYS, QSYS2, SYSPROC, and SYSIBMADM do not need to be specified. If not included in the path, they are implicitly assumed as the last schemas (in this case, it is not included in the CURRENT PATH special register).

The initial value of the CURRENT PATH special register is *LIBL if system naming was used for the first SQL statement run in the activation group. The initial value is "QSYS","QSYS2","SYSPROC","SYSIBMADM","X" (where X is the value of the USER special register) if SQL naming was used for the first SQL statement.

Using the SQL path: The CURRENT PATH special register is used to resolve user-defined distinct types, functions, and procedures in dynamic SQL statements. For more information see “SQL path” on page 63.

Example

The following statement sets the CURRENT PATH special register.

```
SET PATH = FERMAT, "McDuff", SYSIBM
```

The following statement retrieves the current value of the SQL path special register into the host variable called CURPATH.

```
EXEC SQL VALUES (CURRENT PATH) INTO :CURPATH;
```

The value would be "FERMAT","McDuff","SYSIBM" if set by the previous example.

SET RESULT SETS

The SET RESULT SETS statement specifies the result sets that can be returned from a procedure.

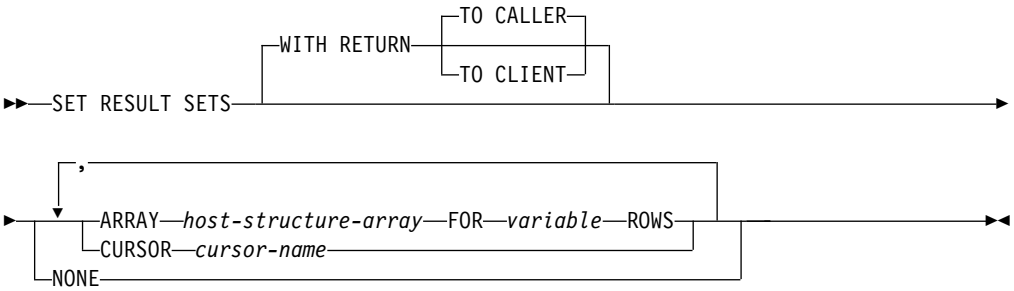
Invocation

This statement can only be embedded in an application program or SQL procedure. It is an executable statement that cannot be dynamically prepared. It is not allowed in a Java or REXX procedure.

Authorization

None required.

Syntax



Description

WITH RETURN
Specifies that the result table of the cursor is intended to be used as a result set that will be returned from a procedure.

For non-scrollable cursors, the result set consists of all rows from the current cursor position to the end of the result table. For scrollable cursors, the result set consists of all rows of the result table.

TO CALLER
Specifies that the cursor can return a result set to the caller of the procedure. For example, if the caller is a client application, the result set is returned to the client application.

TO CLIENT
Specifies that the cursor can return a result set to the client application. This cursor is invisible to any intermediate nested procedures. If a function or trigger called the procedure either directly or indirectly, result sets cannot be returned to the client and the cursor will be closed after the procedure finishes.

CURSOR *cursor-name*
Identifies a cursor to be used to define a result set that can be returned from a procedure. The *cursor-name* must identify a declared cursor as explained in "Description" on page 1080 for the DECLARE CURSOR statement. When the SET RESULT SETS statement is executed, the cursor must be in the open state. It cannot be an allocated cursor.

ARRAY *host-structure-array*
host-structure-array identifies an array of host structures defined in accordance with the rules for declaring host structures. The array cannot contain a C NUL-terminated host variable.

The first structure in the array corresponds to the first row of the result set, the second structure in the array corresponds to the second row of the result set, and so on. In addition, the first value in the row corresponds to the first item in the structure, the second value in the row corresponds to the second item in the structure, and so on.

LOBs and XML cannot be returned in an array when using DRDA.

Only one array can be specified in a SET RESULT SETS statement, including any RETURN TO CLIENT array result sets from nested calls to procedures.

FOR *variable* **ROWS**
Specifies the number of rows in the result set. The *variable* must be a numeric variable with zero scale, and it must not include an indicator variable. It must not be a global variable. The number of rows specified must be in the range of 0 to 32767 and must be less than or equal to the dimension of the host structure array.

NONE
Specifies that no result sets will be returned. Cursors left open when the procedure ends will not be returned.

Notes

For more information about result sets, see Result sets from procedures and WITH RETURN clause.

External procedures: There are three ways to return result sets from an external procedure:

- If a SET RESULT SETS statement is executed in the procedure, the SET RESULT SETS statement identifies the result sets. The result sets are returned in the order specified on the SET RESULT SETS statement.
- If a SET RESULT SETS statement is not executed in the procedure,
  - If no cursors have specified a WITH RETURN clause, each cursor that the procedure opens and leaves open when it returns identifies a result set. The result sets are returned in the order in which the cursors are opened.
  - If any cursors have specified a WITH RETURN clause, each cursor that is defined with the WITH RETURN clause that the procedure opens and leaves open when it returns identifies a result set. The result sets are returned in the order in which the cursors are opened.

When a result set is returned using an open cursor, the rows are returned starting with the current cursor position.

The RESULT SETS clause should be specified on the ALTER PROCEDURE (External), CREATE PROCEDURE (External) statement, or DECLARE PROCEDURE statement to return result sets from a procedure. The maximum number of result sets returned cannot be larger than the number specified on the ALTER PROCEDURE (External), CREATE PROCEDURE (External) statement, or DECLARE PROCEDURE statement.

SQL procedures: In order to return result sets from an SQL procedure, the procedure must be created with the RESULT SETS clause. Each cursor that is defined with the WITH RETURN clause that the procedure opens and leaves open when it returns identifies a result set.

- If a SET RESULT SETS statement is executed in the procedure, the SET RESULT SETS statement identifies which of these result sets to return. The result sets are returned in the order specified on the SET RESULT SETS statement.
- If a SET RESULT SETS statement is not executed in the procedure the result sets are returned in the order in which the cursors are opened.

When a result set is returned using an open cursor, the rows are returned starting with the current cursor position.

The RESULT SETS clause must be specified on the CREATE PROCEDURE (SQL) statement to return any result sets from an SQL procedure. The maximum number of result sets returned cannot be larger than the number specified on the CREATE PROCEDURE statement.

Example

The following SET RESULT SETS statement specifies cursor X as the result set that will be returned when the procedure is called. For more information and complete examples showing the use of result sets from ODBC clients, see the IBM i Access Family topic collection.

```
EXEC SQL SET RESULT SETS CURSOR X;
```

SET SCHEMA

The SET SCHEMA statement changes the value of the CURRENT SCHEMA special register.

Invocation

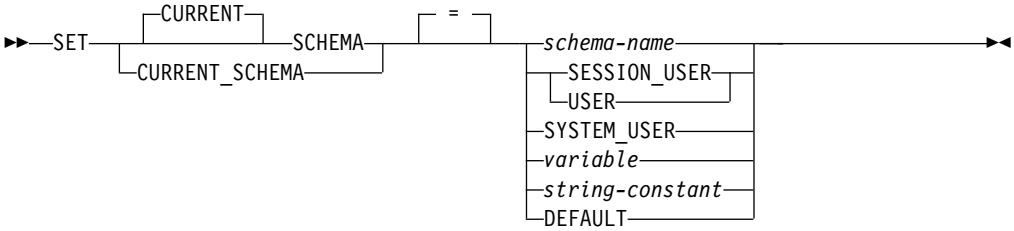
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

If a global variable is referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Description

schema-name

Identifies a schema. If the specified schema name is a system schema name, no validation that the schema exists is made at the time the current schema is set. If the specified schema name is not a system schema name, the schema must exist at the time the current schema is set.

If the value specified does not conform to the rules for a *schema-name*, an error is returned.

SESSION_USER or USER

This value is the SESSION_USER special register.

SYSTEM_USER

This value is the SYSTEM_USER special register.

variable

Specifies a variable which contains a schema name. It can be a global variable if it is qualified with schema name. The content is not folded to uppercase.

The variable:

- Must be a character-string or Unicode graphic variable. The actual length of the contents of the *variable* after trimming any trailing blanks must not exceed the length of a schema name. See Appendix A, “SQL limits,” on page 1493.
- Must not be followed by an indicator variable.

- Must not be the null value.
- Must conform to the rules for forming an ordinary or delimited identifier.
- Must be padded on the right with blanks if the variable is fixed length.
- Must not be the keyword SESSION_USER, SYSTEM_USER, or USER.

string-constant
A character constant with a schema name.

The string constant:

- Must have a length after trimming any trailing blanks that does not exceed the maximum length of a schema name
- Must include a schema name that is left justified and conforms to the rules for forming an ordinary or delimited identifier.
- Must not be the keyword SESSION_USER, SYSTEM_USER, or USER.

DEFAULT
The CURRENT SCHEMA is set to its initial value. The initial value for SQL naming is USER. The initial value for system naming is *LIBL.

Notes

Considerations for keywords: There is a difference between specifying a single keyword (such as USER) as a single keyword or as a delimited identifier. To indicate that the current value of the USER special register should be used for setting the current schema, specify USER as a keyword. If USER is specified as a delimited identifier instead (for example, "USER"), it is interpreted as a schema name of that value ("USER").

Transaction considerations: The SET SCHEMA statement is not a committable operation. ROLLBACK has no effect on the CURRENT SCHEMA.

Impact on other special registers: Setting the CURRENT SCHEMA special register does not effect the CURRENT PATH special register. Hence, the CURRENT SCHEMA will not be included in the SQL path and functions, procedures and user-defined type resolution may not find these objects. To include the current schema value in the SQL path, whenever the SET SCHEMA statement is issued, also issue the SET PATH statement including the schema name from the SET SCHEMA statement.

CURRENT SCHEMA: The value of the CURRENT SCHEMA special register is used as the qualifier for some unqualified names in all dynamic SQL statements except in programs where the DYNDFTCOL has been specified. If DYNDFTCOL is specified in a program, its schema name is used instead of the CURRENT SCHEMA schema name. For information about qualification of names, see “Qualification of unqualified object names” on page 63.

For SQL naming, the initial value of the CURRENT SCHEMA special register is equivalent to USER. For system naming, the initial value of the CURRENT SCHEMA special register is '*LIBL'.

Syntax alternatives: CURRENT SQLID is accepted as a synonym for CURRENT SCHEMA and the effect of a SET CURRENT SQLID statement will be identical to that of a SET CURRENT SCHEMA statement. No other effects, such as statement authorization changes, will occur.

SET SCHEMA is equivalent to calling the QSQCHGDC API.

Examples

Example 1: The following statement sets the CURRENT SCHEMA special register.

```
SET SCHEMA = RICK
```

Example 2: The following example retrieves the current value of the CURRENT SCHEMA special register into the host variable called CURSCHEMA.

```
EXEC SQL VALUES(CURRENT SCHEMA) INTO :CURSCHEMA
```

The value would be RICK, set by the previous example.

SET SESSION AUTHORIZATION

The SET SESSION AUTHORIZATION statement changes the value of the SESSION_USER and USER special registers. It also changes the name of the user profile associated with the current thread.

Invocation

This statement can be embedded within an application program or issued interactively. It is an executable statement that can be dynamically prepared. It must not be specified in REXX.

SET SESSION AUTHORIZATION is not allowed in an SQL trigger, SQL function, or SQL procedure.

Authorization

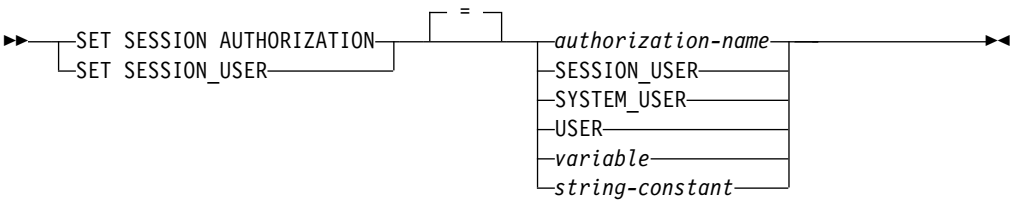
If the authorization name specified on the statement is different than the value in the SYSTEM_USER special register, the privileges held by the authorization ID of the statement must include the system authority of *ALLOBJ.

No authorization is required to execute this statement if the authorization name specified on the statement is the same as the SYSTEM_USER special register.

If a global variable is referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Description

authorization-name
Identifies an authorization ID that is to be used as the new value for the SESSION_USER special register and the runtime authorization ID.

The authorization ID must be a valid user profile or group user profile that exists at the current server. Several system user profiles that do not have a user profile handle may not be used. For more information, see the Get Profile Handle API.

SESSION_USER or USER
The SESSION_USER special register and the runtime authorization ID are set to the USER special register.

SYSTEM_USER
The SESSION_USER special register and the runtime authorization ID are set to the SYSTEM_USER special register.

variable
A variable which contains an authorization ID name. It can be a global variable if it is qualified with schema name.

The variable:

- Must be a character-string variable.
- If *variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value
- Must include an authorization ID that is left justified and must conform to the rules for forming an ordinary or delimited identifier.
- Must be padded on the right with blanks.
- Must not be the null value.
- Must not be the keyword USER, SESSION_USER, or SYSTEM_USER.

string-constant
A character constant with an authorization ID.

Notes

Other effects of SET SESSION AUTHORIZATION: SET SESSION AUTHORIZATION causes the following to occur:

- All cursors that were opened during the unit of work are closed.
- All LOB locators are freed.
- All locks acquired under this unit of work's commitment definition are released.
- All prepared statements are destroyed.
- All SQL descriptor areas are deallocated.
- All procedure result sets are cleared.
- The encryption password is reset.
- All open native database files and Integrated File System (IFS) files are closed, including sockets, NTC sessions, and memory maps.

Other resources are preserved when SET SESSION AUTHORIZATION is executed, including global variables and declared temporary tables. It is recommended that all declared temporary tables be dropped or cleared and global variables be cleared before executing the SET SESSION AUTHORIZATION statement.

SET SESSION AUTHORIZATION restrictions: This statement can only be issued as the first statement that results in work that might be backed out during the unit of work. The following executable statements may be issued prior to executing SET SESSION AUTHORIZATION:

- All SQL transaction statements
- All SQL connection statements
- All SQL session statements
- GET DIAGNOSTICS

SET SESSION AUTHORIZATION is not allowed if any connections other than the connection to the current server exist, including any local connections to a non-default activation group.

SET SESSION AUTHORIZATION is not allowed if any held cursors are open or any held locators exist.

SET SESSION AUTHORIZATION scope: The scope of the SET SESSION AUTHORIZATION is the current thread. Other threads in the application process are unaffected.

Examples

Example 1: The following statement sets the SESSION_USER special register.

```
SET SESSION_USER = RAJIV
```

Example 2: Set the session authorization ID (the SESSION_USER special register) to be the value of the system authorization ID, which is the ID that established the connection on which the statement has been issued.

```
SET SESSION AUTHORIZATION SYSTEM_USER
```

SET TRANSACTION

The SET TRANSACTION statement sets the isolation level, read only attribute, or diagnostics area size for the current unit of work.

Invocation

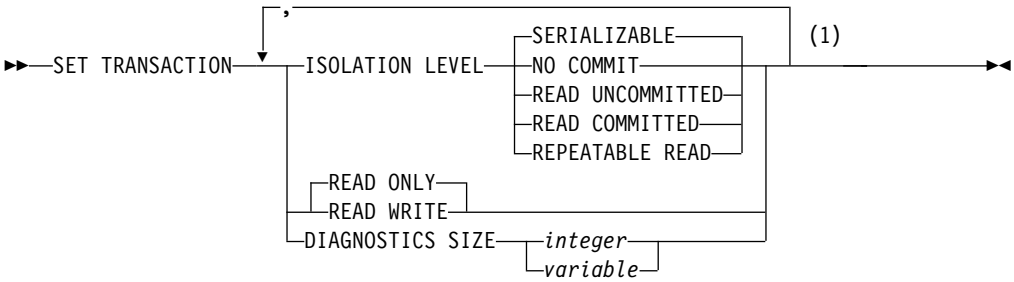
This statement can be embedded within an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

If a global variable is referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the global variable identified in the statement,
  - The READ privilege on the global variable, and
  - The system authority *EXECUTE on the library containing the global variable
- Administrative authority

Syntax



Notes:

1 Only one ISOLATION LEVEL clause, one READ WRITE or READ ONLY clause, and one DIAGNOSTICS SIZE clause may be specified.

Description

ISOLATION LEVEL
Specifies the isolation level of the transaction. If the ISOLATION LEVEL clause is not specified, ISOLATION LEVEL SERIALIZABLE is implicit

NO COMMIT
Specifies isolation level NC (COMMIT(*NONE)).

READ UNCOMMITTED
Specifies isolation level UR (COMMIT(*CHG)).

READ COMMITTED
Specifies isolation level CS (COMMIT(*CS)).

REPEATABLE READ
Specifies isolation level RS (COMMIT(*ALL)).

SERIALIZABLE
Specifies isolation level RR (COMMIT(*RR)).

READ WRITE or READ ONLY
Specifies whether the transaction allows data change operations.

READ WRITE
Specifies that all SQL operations are allowed. This is the default unless ISOLATION LEVEL READ UNCOMMITTED is specified.

READ ONLY
Specifies that only SQL operations that do not change SQL data are allowed. If ISOLATION LEVEL READ UNCOMMITTED is specified, this is the default.

DIAGNOSTICS SIZE
Specifies the maximum number of GET DIAGNOSTICS condition areas for the current transaction. The GET DIAGNOSTICS *statement-information-item* MORE will be set to 'Y' for the current statement if the statement exceeds the maximum number of condition areas for the current transaction. The maximum size of the diagnostics area is 90K. The specified maximum number of condition areas must be between 1 and 32767.

integer
An integer constant that specifies the maximum number of condition areas for the current transaction.

variable
Identifies a variable which contains the maximum number of condition areas for the current transaction. The variable must be a numeric variable with a zero scale and must not be followed by an indicator variable.

Notes

Scope of SET TRANSACTION: The SET TRANSACTION statement sets the isolation level for SQL statements for the current activation group of the process. If that activation group has commitment control scoped to the job, then the SET TRANSACTION statement sets the isolation level of all other activation groups with job commit scoping as well.

If an isolation clause is specified in an SQL statement, that isolation level overrides the transaction isolation level and is used for dynamic SQL statements.

The scope of the SET TRANSACTION statement is based on the context in which it is run. If the SET TRANSACTION statement is run in a trigger, the isolation level specified applies to all subsequent SQL statements until another SET TRANSACTION statement occurs or until the trigger completes, whichever happens first. If the SET TRANSACTION statement is run outside a trigger, the isolation level specified applies to all subsequent SQL statements (except those statements within a trigger that are executed after a SET TRANSACTION statement in the trigger) until a COMMIT or ROLLBACK operation occurs.

For more information about isolation levels, see “Isolation level” on page 26.

SET TRANSACTION restrictions: The SET TRANSACTION statement can only be executed when it is the first SQL statement in a unit of work, unless:

- all previous statements executed in the unit of work are SET TRANSACTION statements or statements that are executed under isolation level NC, or
- it is executed in a trigger.

In a trigger, SET TRANSACTION with READ ONLY is allowed only on a COMMIT boundary. The SET TRANSACTION statement can be executed in a trigger at any time, but it is recommended that it be executed as the first statement in the trigger. The SET TRANSACTION statement is useful within triggers to set the isolation level for SQL statements in the trigger to the same level as the application which caused the trigger to be activated.

A SET TRANSACTION statement is not allowed if the current connection is to a remote application server unless it is in a trigger at the current server. Once a SET TRANSACTION statement is executed, CONNECT and SET CONNECTION statements are not allowed until the unit of work is committed or rolled back.

SET TRANSACTION is not allowed as the first statement in a secondary thread.

The SET TRANSACTION statement has no effect on WITH HOLD cursors that are still open when the SET TRANSACTION statement is executed.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords NC or NONE can be used as synonyms for NO COMMIT.
- The keywords UR and CHG can be used as synonyms for READ UNCOMMITTED.
- The keyword CS can be used as a synonym for READ COMMITTED.
- The keywords RS or ALL can be used as synonyms for REPEATABLE READ.
- The keyword RR can be used as a synonym for SERIALIZABLE.

Examples

Example 1: The following SET TRANSACTION statement sets the isolation level to NONE (equivalent to specifying *NONE on the SQL precompiler command).

```
EXEC SQL SET TRANSACTION ISOLATION LEVEL NO COMMIT;
```

Example 2: The following SET TRANSACTION statement sets the isolation level to SERIALIZABLE.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

SET transition-variable

The SET transition-variable statement assigns values to new transition variables.

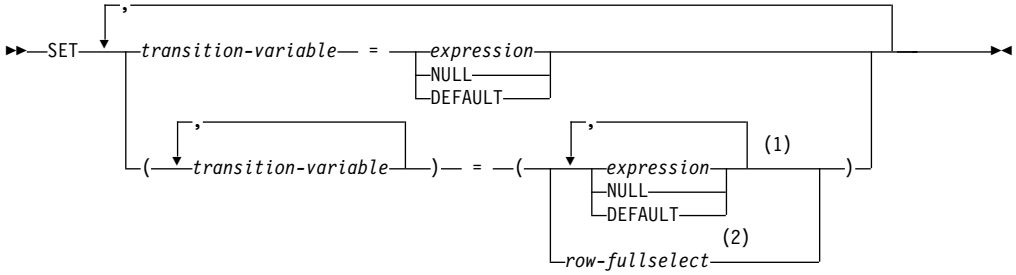
Invocation

This statement can only be used as an SQL statement in a BEFORE trigger. It is an executable statement that cannot be dynamically prepared.

Authorization

If a *row-fullselect* is specified, see "fullselect" on page 676 for an explanation of the authorization required for each subselect.

Syntax



Notes:

1 The number of *expression*s, NULLs, and DEFAULTs must match the number of *transition-variable*s.

2 The number of columns in the select list must match the number of *transition-variable*s.

Description

transition-variable
Identifies the column to be updated in the new row. A *transition-variable* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value. An OLD *transition-variable* must not be identified.

A *transition-variable* must not be identified more than once in the same SET transition-variable statement.

The data type of each *transition-variable* must be compatible with its corresponding result column. Values are assigned to *transition-variable*s according to the storage assignment rules. For more information see "Assignments and comparisons" on page 98.

expression
Specifies the new value of the *transition-variable*. The *expression* is any expression of the type described in "Expressions" on page 165. The expression cannot include an aggregate function.

An *expression* may contain references to OLD and NEW *transition-variable*s. If the CREATE TRIGGER statement contains both OLD and NEW clauses, references to *transition-variable*s must be qualified by the *correlation-name*.

NULL
Specifies the null value. NULL can only be specified for nullable columns.

DEFAULT
Specifies that the default value of the column associated with the *transition-variable* will be used. DEFAULT is not allowed if the column is an IDENTITY column, a row change timestamp column, or has a ROWID data type.

row-fullselect
A fullselect that returns a single result row. The result column values are assigned to each corresponding *transition-variable*. If the result of the fullselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

Notes

Multiple assignments: If more than one assignment is included in the same SET *transition-variable* statement, all *expression*s are evaluated before the assignments are performed. Thus, references to *transition-variable*s in an expression are always the value of the *transition-variable*s prior to any assignment in the SET statement.

Examples

Example 1: Ensure that the salary column is never greater than 50000. If the new value is greater than 50000, set it to 50000.

```
CREATE TRIGGER LIMIT_SALARY
  BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AS NEW_VAR
  FOR EACH ROW MODE DB2SQL
  WHEN (NEW_VAR.SALARY > 50000)
    BEGIN ATOMIC
    SET NEW_VAR.SALARY = 50000;
  END
```

Example 2: When the job title is updated, increase the salary based on the new job title. Assign the years in the position to 0.

```
CREATE TRIGGER SET_SALARY
  BEFORE UPDATE OF JOB ON STAFF
  REFERENCING OLD AS OLD_VAR
              NEW AS NEW_VAR
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET (NEW_VAR.SALARY, NEW_VAR.YEARS) =
        (OLD_VAR.SALARY * CASE NEW_VAR.JOB
            WHEN 'Sales' THEN 1.1
            WHEN 'Mgr'   THEN 1.05
            ELSE 1 END ,0);
  END
```

SET variable

The SET variable statement produces a result table consisting of at most one row and assigns the values in that row to variables.

Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared if the all variables being set are global variables. It must not be specified in REXX.

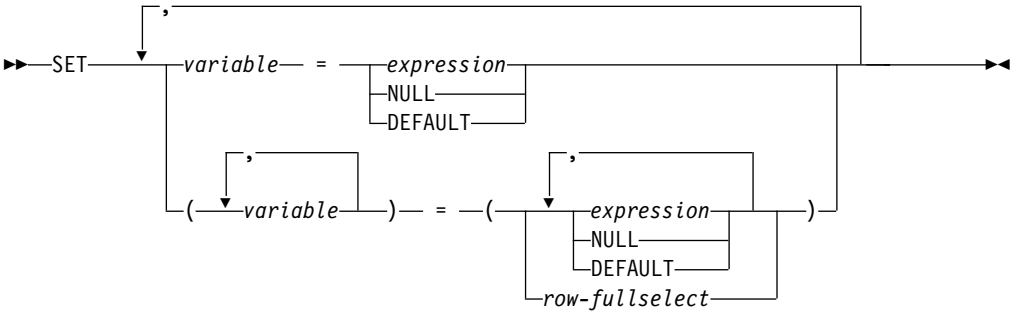
Authorization

If a *row-fullselect* is specified, see "fullselect" on page 676 for an explanation of the authorization required for each subselect.

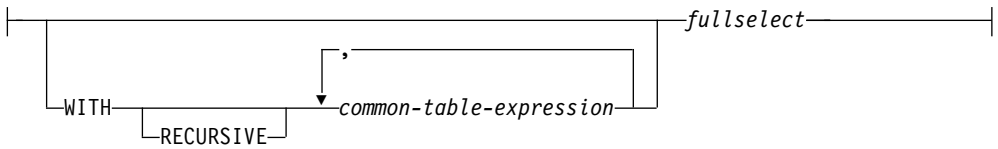
If a global variable is specified on the left hand side of the assignment, the privileges held by the authorization ID of the statement must include:

- The WRITE privilege on the global variable.

Syntax



row-fullselect:



Description

variable, ...

Identifies one or more variables or host structures that must be declared in accordance with the rules for declaring variables (see "References to host variables" on page 149). A host structure is logically replaced by a list of variables that represent each of the elements of the host structure.

The value to be assigned to each *variable* can be specified immediately following the *variable*, for example, *variable* = *expression*, *variable* = *expression*. Or, sets of parentheses can be used to specify all the *variables* and then all the values, for example, (*variable*, *variable*) = (*expression*, *expression*).

The data type of each variable must be compatible with its corresponding result column. Each assignment is made according to the rules described in "Assignments and comparisons" on page 98. The number of *variables* specified to the left of the equal operator must equal the number of values in the corresponding result specified to the right of the equal operator. If the value is null, an indicator variable must be provided. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

If an error occurs as the result of an arithmetic expression in the *expression* or SELECT list of the subselect (division by zero, or overflow) or a character conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided. The value of the variable is undefined. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, a warning is returned.) If you do not provide an indicator variable, an error is returned. It is possible that some values have already been assigned to variables and will remain assigned when the error occurs.

expression
Specifies the new value of the variable. The *expression* is any expression of the type described in "Expressions" on page 165. It must not include a column name.

NULL
Specifies that the new value for the variable is the null value.

DEFAULT
Specifies that the new value for the variable is its initial default value. DEFAULT can only be assigned to a global variable. Only one variable can be assigned in the SET statement when DEFAULT is used.

row-fullselect
A fullselect that returns a single result row. The result column values are assigned to each corresponding *variable*. If the result of the fullselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

WITH *common-table-expression*
Specifies a common table expression. For an explanation of common table expression, see "common-table-expression" on page 683.

fullselect
A *fullselect* that returns a single result row. The result column values are assigned to each corresponding *variable*. If the result of the *fullselect* is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

Notes

Variable assignment: If the specified variable is character and is not large enough to contain the result, a warning (SQLSTATE 01004) is returned (and 'W' is assigned to SQLWARN1 in the SQLCA). The actual length of the result is returned in the indicator variable associated with the variable, if an indicator variable is provided.

If the specified variable is a C NUL-terminated variable and is not large enough to contain the result and the NUL-terminator:

- If the *CNULRQD option is specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*YES) on the SET OPTION statement), the following occurs:
  - The result is truncated.
  - The last character is the NUL-terminator.
  - The value 'W' is assigned to SQLWARN1 in the SQLCA.
- If the *NOCNULRQD option on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement) is specified, the following occurs:
  - The NUL-terminator is not returned.
  - The value 'N' is assigned to SQLWARN1 in the SQLCA.

Multiple assignments: If more than one assignment is included in the same SET statement, all *expression*s and *row-fullselect*s are completely evaluated before the assignments are performed. Thus, references to a target variable in an *expression* or *row-fullselect* are always the value of the target variable prior to any assignment in the SET statement.

Examples

Example 1: Assign the value of the CURRENT PATH special register to host variable HV1.

```
EXEC SQL SET :HV1 = CURRENT PATH;
```

Example 2: Assume that LOB locator LOB1 is associated with a CLOB value. Assign a portion of the CLOB value to host variable DETAILS using the LOB locator.

```
EXEC SQL SET :DETAILS = SUBSTR(:LOB1,1,35);
```

SIGNAL

The SIGNAL statement signals an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE and optional *condition-information-items*.

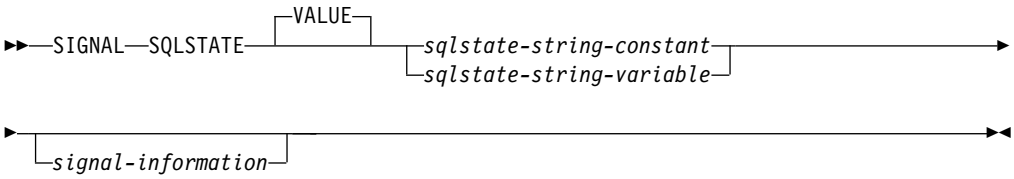
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

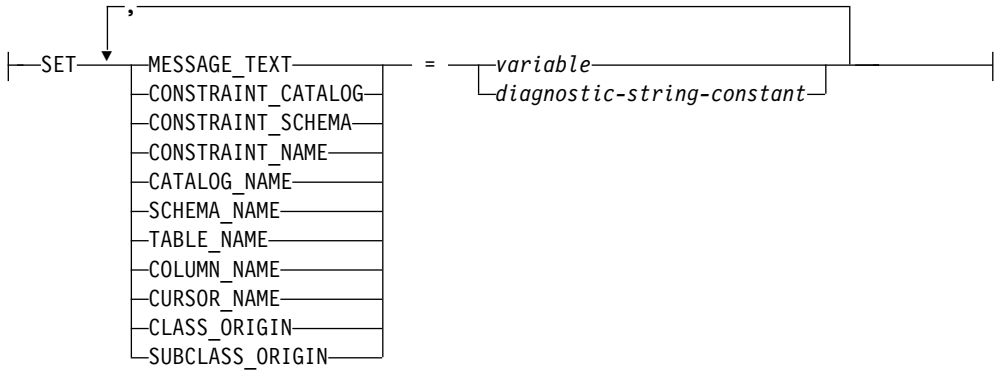
Authorization

None required.

Syntax



signal-information:



Description

SQLSTATE VALUE

Specifies the SQLSTATE that will be signalled. The specified value must not be null and must follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00' since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is returned.

sqlstate-string-constant
The *sqlstate-string-constant* must be a character string constant with exactly 5 characters.

sqlstate-string-variable
The *sqlstate-string-variable* must be a character or Unicode graphic variable. It cannot be a global variable. The actual length of the contents of the *variable* must be 5.

SET
Introduces the assignment of values to *condition-information-items*. The *condition-information-item* values can be accessed using the GET DIAGNOSTICS statement. The only *condition-information-item* that can be accessed in the SQLCA is MESSAGE_TEXT.

MESSAGE_TEXT
Specifies a string that describes the error or warning.

If an SQLCA is used,

- the string is returned in the SQLERRMC field of the SQLCA
- if the actual length of the string is longer than 1000 bytes, it is truncated without a warning.

CONSTRAINT_CATALOG
Specifies a string that indicates the name of the database that contains a constraint related to the signalled error or warning.

CONSTRAINT_SCHEMA
Specifies a string that indicates the name of the schema that contains a constraint related to the signalled error or warning.

CONSTRAINT_NAME
Specifies a string that indicates the name of a constraint related to the signalled error or warning.

CATALOG_NAME
Specifies a string that indicates the name of the database that contains a table or view related to the signalled error or warning.

SCHEMA_NAME
Specifies a string that indicates the name of the schema that contains a table or view related to the signalled error or warning.

TABLE_NAME
Specifies a string that indicates the name of a table or view related to the signalled error or warning.

COLUMN_NAME
Specifies a string that indicates the name of a column in the table or view related to the signalled error or warning.

CURSOR_NAME
Specifies a string that indicates the name of a cursor related to the signalled error or warning.

CLASS_ORIGIN
Specifies a string that indicates the origin of the SQLSTATE class related to the signalled error or warning.

SUBCLASS_ORIGIN
Specifies a string that indicates the origin of the SQLSTATE subclass related to the signalled error or warning.

variable
Identifies a variable that must be declared in accordance with the rules for declaring variables (see "References to host variables" on page 149). The variable contains the value to be assigned to the *condition-information-item*. The variable must be defined as CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable. It cannot be a global variable.

diagnostic-string-constant
Specifies a character string constant that contains the value to be assigned to the *condition-information-item*.

Notes

SQLSTATE values: Any valid SQLSTATE value can be used in the SIGNAL statement. However, it is recommended that programmers define new SQLSTATEs based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

SQLSTATE values are comprised of a two-character class code value, followed by a three-character subclass code value. Class code values represent classes of successful and unsuccessful execution conditions.

- SQLSTATE classes that begin with the characters '7' through '9' or 'I' through 'Z' may be defined. Within these classes, any subclass may be defined.
- SQLSTATE classes that begin with the characters '0' through '6' or 'A' through 'H' are reserved for the database manager. Within these classes, subclasses that begin with the characters '0' through 'H' are reserved for the database manager. Subclasses that begin with the characters 'I' through 'Z' may be defined.

For more information about SQLSTATEs, see the SQL Messages and Codes topic collection.

Assignment: When the SIGNAL statement is executed, the value of each of the specified *string-constants* and *variables* is assigned to the corresponding *condition-information-item*. However, if the length of a *string-constant* or *variable* is longer than the maximum length of the corresponding *condition-information-item*, it is truncated without a warning. For details on the assignment rules, see "Assignments and comparisons" on page 98. For details on the maximum length of specific *condition-information-items*, see "GET DIAGNOSTICS" on page 1194.

Processing a SIGNAL statement: When a SIGNAL statement is issued, the SQLCODE is based on the SQLSTATE value as follows:

- If the specified SQLSTATE class is either '01' or '02', a warning or not found is signalled and the SQLCODE is set to +438.
- Otherwise, an exception is signalled and the SQLCODE is set to –438.

Examples

Example 1: Signal SQLSTATE '75002' with a descriptive message text.

```
EXEC SQL SIGNAL SQLSTATE '75002'
           SET MESSAGE_TEXT = 'Customer number is not known';
```

Example 2: Signal SQLSTATE '75002' with a descriptive message text and associate a specific table with the error.

```
EXEC SQL SIGNAL SQLSTATE '75002'
             SET MESSAGE_TEXT = 'Customer number is not known',
                 SCHEMA_NAME = 'CORPDATA',
                 TABLE_NAME = 'CUSTOMER';
```

UPDATE

The UPDATE statement updates the values of specified columns in rows of a table or view. Updating a row of a view updates a row of its base table, if no INSTEAD OF UPDATE trigger is defined on this view. If such a trigger is defined, the trigger will be activated instead.

There are two forms of this statement:

- The *Searched* UPDATE form is used to update one or more rows (optionally determined by a search condition).
- The *Positioned* UPDATE form is used to update exactly one row (as determined by the current position of a cursor).

Invocation

A Searched UPDATE statement can be embedded in an application program or issued interactively. A Positioned UPDATE must be embedded in an application program. Both forms are executable statements that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table or view identified in the statement:
  - The UPDATE privilege on the table or view, or
  - The UPDATE privilege on each column to be updated; and
  - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

If the *expression* in the *assignment-clause* contains a reference to a column of the table or view, or if the *search-condition* in a Searched UPDATE contains a reference to a column of the table or view, then the privileges held by the authorization ID of the statement must also include one of the following:

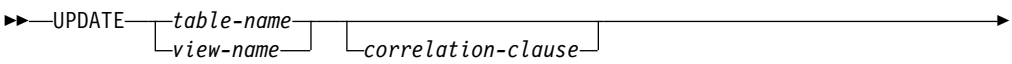
- The SELECT privilege on the table or view
- Administrative authority

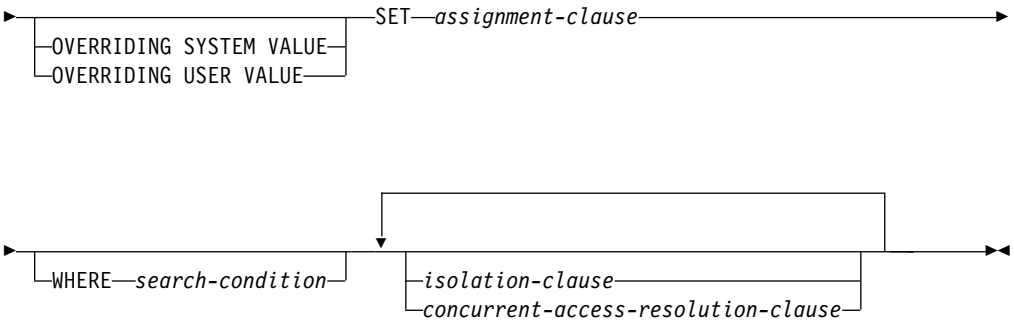
If the *search-condition* includes a subquery or if the *assignment-clause* includes a *scalar-fullselect* or *row-fullselect*, see Chapter 5, “Queries,” on page 627 for an explanation of the authorization required for each subselect.

For information about the system authorities corresponding to SQL privileges, see Corresponding System Authorities When Checking Privileges to a Table or View.

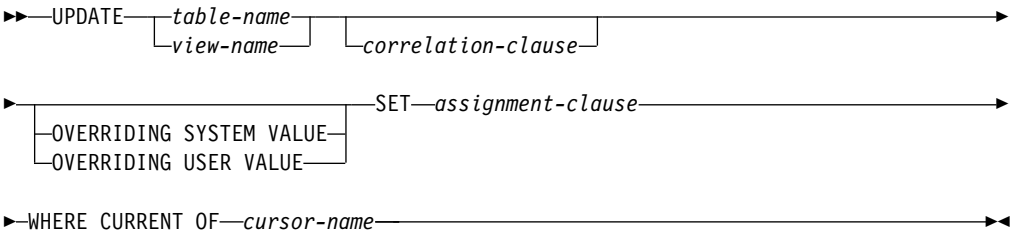
Syntax

Searched UPDATE:

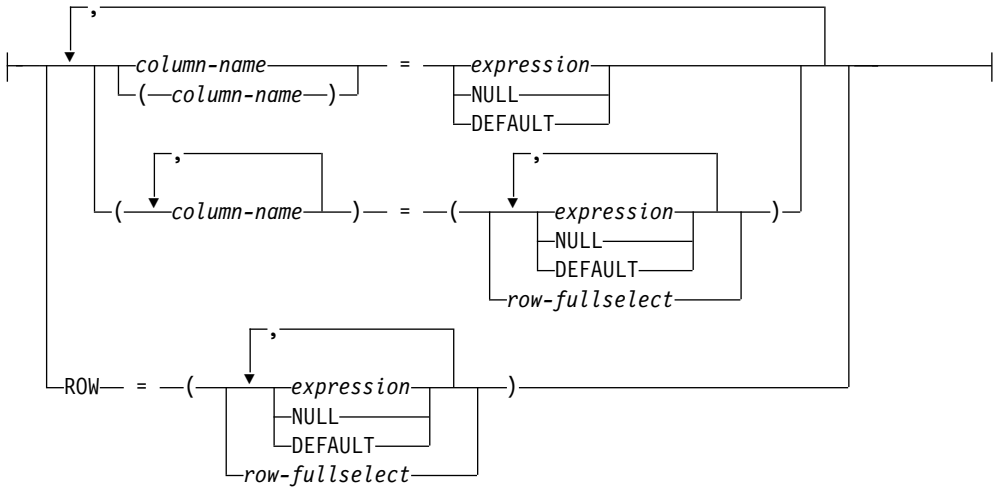




Positioned UPDATE:



assignment-clause:



isolation–clause:

Description

table-name **or** *view-name*
Identifies the table or view to be updated. The name must identify a table or view that exists at the current server, but it must not identify a catalog table, a

view of a catalog table, or a read-only view. For an explanation of read-only views and updatable views, see "CREATE VIEW" on page 1069.

correlation-clause
Can be used within *search-condition* or *assignment-clause* to designate the table or view. For an explanation of *correlation-clause*, see "table-reference" on page 633. For an explanation of *correlation-name*, see "Correlation names" on page 140.

OVERRIDING SYSTEM VALUE or OVERRIDING USER VALUE
Specifies whether system-generated values or user-specified values for a ROWID, identity, or row change timestamp column are used. If OVERRIDING SYSTEM VALUE is specified, the implicit or explicit list of columns in the SET clause must contain a column defined as GENERATED ALWAYS. If OVERRIDING USER VALUE is specified, the implicit or explicit list of columns in the SET clause must contain a column defined as either GENERATED ALWAYS or GENERATED BY DEFAULT.

OVERRIDING SYSTEM VALUE
Specifies that the value specified in the SET clause for a column that is defined as GENERATED ALWAYS is used. A system-generated value is not used.

OVERRIDING USER VALUE
Specifies that the value specified in the SET clause for a column that is defined as either GENERATED ALWAYS or GENERATED BY DEFAULT is ignored. Instead, a system-generated value is used, overriding the user-specified value.

If neither OVERRIDING SYSTEM VALUE or OVERRIDING USER VALUE is specified:

- A value cannot be specified for a ROWID, identity, or row change timestamp column that is defined as GENERATED ALWAYS.
- A value can be specified for a ROWID column that is defined as GENERATED BY DEFAULT. If a value is specified, that value is assigned to the column. However, a value in a ROWID column defined BY DEFAULT can be updated only if the specified value is a valid row ID value that was previously generated by DB2 for z/OS or DB2 for i.
- A value can be specified for an identity or row change timestamp column that is defined as GENERATED BY DEFAULT. When a value of an identity column or row change timestamp column defined BY DEFAULT is updated, the database manager does not verify that the specified value is a unique value for the column unless the identity column or row change timestamp column is the sole key in a unique constraint or unique index. Without a unique constraint or unique index, the database manager can guarantee unique values only among the set of system-generated values as long as NO CYCLE is in effect.

  If a value is not specified the database manager generates a new value.

SET
Introduces the assignment of values to column names.

assignment-clause

column-name
Identifies a column to be updated. The *column-name* must identify a column of the specified table or view. If extended indicator variables are

not enabled, that column must not identify a view column derived from a scalar function, constant, or expression. A column must not be specified more than once.

For a Positioned UPDATE:

- If the UPDATE clause was specified in the SELECT statement of the cursor, each column name in the SET list must also appear in the UPDATE clause.
- If the UPDATE clause was not specified in the SELECT statement of the cursor, the name of any updatable column may be specified.

For more information, see "update-clause" on page 690.

A view column derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same UPDATE statement.

If a list of *column-names* is specified, the number of *expressions*, NULLs, and DEFAULTS must match the number of *column-names*.

ROW

Identifies all the columns of the specified table or view except for columns defined with the hidden attribute. If a view is specified, none of the columns of the view may be derived from a scalar function, constant, or expression.

The number of *expressions*, NULLs, and DEFAULTs (or the number of result columns from a *row-fullselect*) must match the number of columns in the row.

For a Positioned UPDATE, if the UPDATE clause was specified in the SELECT statement of the cursor, each column of the table or view must also appear in the UPDATE clause. For more information, see "update-clause" on page 690.

ROW may not be specified for a view that contains a view column derived from the same column as another column of the view, because both columns cannot be updated in the same UPDATE statement.

expression

Specifies the new value of the column. The *expression* is any expression of the type described in "Expressions" on page 165. It must not include an aggregate function.

A *column-name* in an expression must name a column of the named table or view. For each row updated, the value of the column in the expression is the value of the column in the row before the row is updated.

Each variable in the clause must identify a host structure or variable that is declared in accordance with the rules for declaring host structures and variables. In the operational form of the statement, a reference to a host structure is replaced by a reference to each of its variables. If *expression* is a single host variable, the host variable can include an indicator that is enabled for extended indicator variables. If extended indicator variables are enabled and an expression in the assignment clause is not a single host variable, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used. For further information about variables and structures, see "References to host variables" on page 149 and "Host structures" on page 155. If a host structure is specified, the keyword ROW must be specified.

NULL
Specifies the new value for a column is the null value. NULL should only be specified for nullable columns.

DEFAULT
Specifies that the default value is assigned to a column. The value that is used depends on how the column was defined, as follows:

- If the column is a ROWID or identity column, the database manager will generate a new value.
- If the WITH DEFAULT clause is used, the default used is as defined for the column (see *default-clause* in *column-definition* in "CREATE TABLE" on page 982).
- If the WITH DEFAULT clause or the NOT NULL clause is not used, the value used is NULL.
- If the column is defined as a row change timestamp column, a new row change timestamp value is assigned to the column.

If the NOT NULL clause is used and the WITH DEFAULT clause is not used, or if DEFAULT NULL is used, the DEFAULT keyword cannot be specified for that column.

DEFAULT must be specified for an identity column defined as GENERATED ALWAYS unless OVERRIDING SYSTEM VALUE is used.

row-fullselect
A fullselect that returns a single result row. The number of result columns in the select list must match the number of *column-names* (or if ROW is specified, the number of columns in the row) specified for assignment. The result column values are assigned to each corresponding *column-name*. If the result of the fullselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

The *row-fullselect* may contain references to columns of the target table of the UPDATE statement. For each row updated, the value of such a column in the expression is the value of the column in the row before the row is updated.

WHERE
Specifies the rows to be updated. The clause can be omitted, or a *search-condition* or *cursor-name* can be specified. If the clause is omitted, all rows of the table or view are updated.

search-condition
Is any search described in "Search conditions" on page 225. Each *column-name* in the search condition, other than in a subquery, must name a column of the table or view. When the search condition includes a subquery in which the same table is the base object of both the UPDATE and the subquery, the subquery is completely evaluated before any rows are updated.

The *search-condition* is applied to each row of the table or view. The updated rows are those for which the results of the *search-condition* are true.

If the *search-condition* contains a subquery, the subquery can be thought of as being executed each time the *search-condition* is applied to a row, and the results of that subquery used in applying the *search-condition*. In actuality, a

subquery with no correlated references may be executed only once. A subquery with a correlated reference may have to be executed once for each row.

CURRENT OF *cursor-name*
Identifies the cursor to be used in the update operation. The *cursor-name* must identify a declared cursor as explained in "DECLARE CURSOR" on page 1079.

The table or view named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. For an explanation of read-only result tables, see "DECLARE CURSOR" on page 1079.

When the UPDATE statement is executed, the cursor must be positioned on a row; that row is updated.

isolation-clause
Specifies the isolation level to be used for this statement.

WITH

Introduces the isolation level, which may be one of:
- *RR* Repeatable read
- *RS* Read stability
- *CS* Cursor stability
- *UR* Uncommitted read
- *NC* No commit

If *isolation-clause* is not specified the default isolation is used. See "isolation-clause" on page 693 for a description of how the default is determined.

concurrent-access-resolution-clause
Specifies the concurrent access resolution to use for the select statement. For more information, see "concurrent-access-resolution-clause" on page 695.

UPDATE Rules

Assignment: Update values are assigned to columns in accordance with the storage assignment rules described in "Assignments and comparisons" on page 98.

Validity: Updates must obey the following rules. If they do not, or if any other errors occur during the execution of the UPDATE statement, no rows are updated.

- *Fullselects:* The *row-fullselect* or *scalar-fullselect* shall return no more than one row (SQLSTATE 21000).
- *Unique constraints and unique indexes:* If the identified table, or the base table of the identified view, has one or more unique indexes or unique constraints, each row update in the table must conform to the limitations imposed by those indexes and constraints (SQLSTATE 23505).

  All uniqueness checks are effectively made at the end of the statement. In the case of a multiple-row UPDATE statement of a column involved in a unique index or unique constraint, this would occur after all rows were updated.
- *Check constraints:* If the identified table, or the base table of the identified view, has one or more check constraints, each check constraint must be true or unknown for each row updated in the table (SQLSTATE 23513).

All check constraints are effectively validated at the end of the statement. In the case of a multiple-row UPDATE statement, this would occur after all rows were updated.

- *Views and the WITH CHECK OPTION:* If a view is identified, the updated rows must conform to any applicable WITH CHECK OPTION (SQLSTATE 44000). For more information, see "CREATE VIEW" on page 1069.

Triggers: If the identified table or the base table of the identified view has an update trigger, the trigger is activated. A trigger might cause other statements to be executed or raise error conditions based on the updated values.

Referential Integrity: The value of the parent key in a parent row must not be changed.

If the update values produce a foreign key that is nonnull, the foreign key must be equal to some value of the parent key of the parent table of the relationship.

The referential constraints (other than a referential constraint with a RESTRICT delete rule) are effectively checked at the end of the statement. In the case of a multiple-row UPDATE statement, this would occur after all rows were updated.

XML values: When an XML column is updated, the new value must be a well-formed XML document.

Extended indicator variable usage: If enabled, indicator variable values other than positive values and 0 (zero) through -7 must not be used. The DEFAULT and UNASSIGNED extended indicator variable values must not appear in contexts in which they are not supported.

Extended indicator variables: In the *assignment-clause* of an UPDATE statement, an expression that is a reference to a single host variable can result in assigning an extended indicator variable value. Assigning an extended indicator variable value of UNASSIGNED has the effect of leaving the target column set to its current values, as if it hadn't be specified in the statement. Assigning an extended indicator variable value of DEFAULT can only be used for columns that have a default value.

If a target column is not updatable, for example an identity column defined as GENERATED ALWAYS, it must be assigned the extended indicator variable value of UNASSIGNED.

An UPDATE statement must not assign all target columns from an extended indicator variable value of UNASSIGNED.

Extended indicator variables and update triggers: If a target column has been assigned an extended indicator variable value of UNASSIGNED, that column is not considered to have been updated.

Extended indicator variables and deferred error checks: When extended indicator variables are enabled, validation that would otherwise be done during statement preparation to recognize an update of a non-updatable column is deferred until the statement is executed.

Notes

Update operation errors: If an update value violates any constraints, or if any other error occurs during the execution of the UPDATE statement and COMMIT(*NONE) was not specified, all changes made during the execution of the statement are backed out. However, other changes in the unit of work made prior to the error are not backed out. If COMMIT(*NONE) is specified, changes are not backed out.

It is possible for an error to occur that makes the state of the cursor unpredictable.

Number of rows updated: When an UPDATE statement completes execution, the number of rows updated is returned in the ROW_COUNT statement information item in the SQL Diagnostics Area (and SQLERRD(3) in the SQLCA). For a description of the SQLCA, see Appendix C, "SQLCA (SQL communication area)," on page 1513.

For a description of ROW_COUNT, see "GET DIAGNOSTICS" on page 1194. For a description of the SQLCA, see Appendix C, "SQLCA (SQL communication area)," on page 1513.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful UPDATE statement. Until these locks are released by a commit or rollback operation, the updated rows can only be accessed by:

- The application process that performed the update.
- Another application process using COMMIT(*NONE) or COMMIT(*CHG) through a read-only operation

The locks can prevent other application processes from performing operations on the table. For further information about locking, see the description of the COMMIT, ROLLBACK, and LOCK TABLE statements, and isolation levels in "Isolation level" on page 26. Also, see the Database Programming topic collection.

A maximum of 500 000 000 rows can be updated or changed in any single UPDATE statement when COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) has been specified. The number of rows changed includes any rows inserted, updated, or deleted under the same commitment definition as a result of a trigger.

REXX: Variables cannot be used in the UPDATE statement within a REXX procedure. Instead, the UPDATE must be the object of a PREPARE and EXECUTE using parameter markers.

Datalinks: If the URL value of a DATALINK column is updated, this is the same as deleting the old DATALINK value then inserting the new one. First, if the old value was linked to a file, that file is unlinked. Then, unless the linkage attributes of the DATALINK value are empty, the specified file is linked to that column.

The comment value of a DATALINK column can be updated without relinking the file by specifying an empty string as the URL path (for example, as the data-location argument of the DLVALUE scalar function or by specifying the new value to be the same as the old value). If a DATALINK column is updated with a null, it is the same as deleting the existing DATALINK value.

An error may occur when attempting to update a DATALINK value if the file server of either the existing value or the new value is no longer registered with the database server

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword NONE can be used as a synonym for NC.
- The keyword CHG can be used as a synonym for UR.
- The keyword ALL can be used as a synonym for RS.

Examples

Example 1: Change the job (JOB) of employee number (EMPNO) '000290' in the EMPLOYEE table to 'LABORER'.

```
UPDATE EMPLOYEE
  SET JOB = 'LABORER'
  WHERE EMPNO = '000290'
```

Example 2: Increase the project staffing (PRSTAFF) by 1.5 for all projects that department (DEPTNO) 'D21' is responsible for in the PROJECT table.

```
UPDATE PROJECT
  SET PRSTAFF = PRSTAFF + 1.5
  WHERE DEPTNO = 'D21'
```

Example 3: All the employees except the manager of department (WORKDEPT) 'E21' have been temporarily reassigned. Indicate this by changing their job (JOB) to NULL and their pay (SALARY, BONUS, COMM) values to zero in the EMPLOYEE table.

```
UPDATE EMPLOYEE
  SET JOB=NULL, SALARY=0, BONUS=0, COMM=0
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'
```

Example 4: In a Java program display the rows from the EMPLOYEE table on the connection context 'ctx' and then, if requested to do so, change the job (JOB) of certain employees to the new job keyed in (NEWJOB).

```
#sql iterator empIterator implements sqlj.runtime.ForUpdate
     with( updateColumns='JOB' )
     ( ... );
empIterator C1;

#sql [ctx] C1 = { SELECT * FROM EMPLOYEE };

#sql { FETCH :C1 INTO ... };
while ( !C1.endFetch() )  {
   System.out.println( ... );
                ...
   if ( condition for updating row ) {
       #sql [ctx] { UPDATE EMPLOYEE
                       SET JOB = :NEWJOB
                       WHERE CURRENT OF :C1 };
   }

   #sql { FETCH :C1 INTO ... };
}
C1.close();
```

VALUES

The VALUES statement provides a method for invoking a user-defined function from a trigger. Transition variables can be passed to the user-defined function.

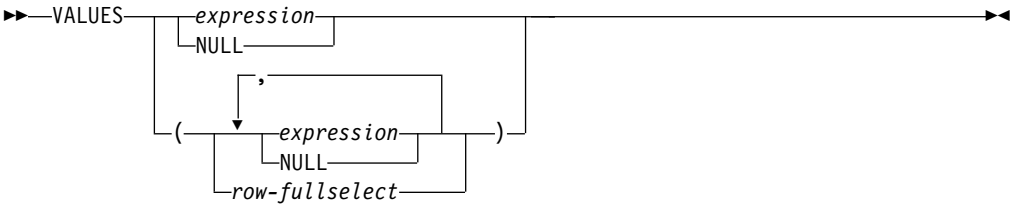
Invocation

This statement can only be used in the triggered action of a CREATE TRIGGER statement.

Authorization

If a *row-fullselect* is specified, see Chapter 5, "Queries," on page 627 for an explanation of the authorization required for each subselect.

Syntax



Description

VALUES
Introduces a single row consisting of one of more columns.

expression
Any expression of the type described in "Expressions" on page 165.

NULL
Specifies the null value.

row-fullselect
A fullselect that returns a single result row. If the result of the fullselect is no rows, then null values are returned. An error is returned if there is more than one row in the result.

Notes

Effects of the statement: The statement is evaluated, but the resulting values are discarded and are not assigned to any output variables. If an error is returned, the database manager stops executing the trigger and rolls back any triggered actions that were performed as well as the statement that caused the triggered action (unless the trigger is running under isolation level *NONE).

Examples

Create an after trigger EMPISRT1 that invokes user-defined function NEWEMP when the trigger is activated. An insert operation on table EMP activates the trigger. Pass transition variables for the new employee number, last name, and first name to the user-defined function.

```
CREATE TRIGGER EMPISRT1
   AFTER INSERT ON EMPLOYEE
   REFERENCING NEW AS N
   FOR EACH ROW
   MODE DB2SQL
   BEGIN ATOMIC
     VALUES( NEWEMP(N.EMPNO, N.LASTNAME, N.FIRSTNAME));
   END
```

VALUES INTO

The VALUES INTO statement produces a result table consisting of at most one row and assigns the values in that row to variables.

Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared, but cannot be issued interactively unless all variables being assigned are global variables. It must not be specified in REXX or Java.

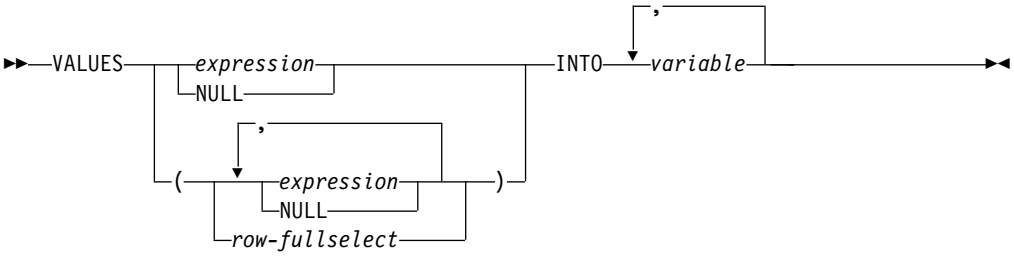
Authorization

If a *row-fullselect* is specified, see Chapter 5, "Queries," on page 627 for an explanation of the authorization required for each subselect.

If a global variable is specified on the right hand side of the assignment, the privileges held by the authorization ID of the statement must include:

- The WRITE privilege on the global variable.

Syntax



Description

VALUES
Introduces a single row consisting of one or more columns.

expression
Specifies the new value of the variable. The *expression* is any expression of the type described in "Expressions" on page 165. It must not include a column name. Host structures are not supported.

NULL
Specifies that the new value for the variable is the null value.

row-fullselect
A fullselect that returns a single result row. The result column values are assigned to each corresponding *variable*. If the result of the fullselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

INTO *variable*,...
Identifies one or more host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and variables. In the operational form of INTO, a reference to a host structure is

replaced by a reference to each of its variables. The first value specified is assigned to the first variable, the second value to the second variable, and so on.

Notes

Variable assignment: Each assignment to a variable is performed according to the retrieval assignment rules described in "Assignments and comparisons" on page 98.[119] If the number of variables is less than the number of values in the row, an SQL warning (SQLSTATE 01503) is returned (and the SQLWARN3 field of the SQLCA is set to 'W'). Note that there is no warning if there are more variables than the number of result columns. If a value is null, an indicator variable must be provided for that value.

If the specified variable is character and is not large enough to contain the result, a warning (SQLSTATE 01004) is returned (and 'W' is assigned to SQLWARN1 in the SQLCA). The actual length of the result may be returned in the indicator variable associated with the variable, if an indicator variable is provided. For further information, see "Variables" on page 147.

If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

If the specified variable is a C NUL-terminated host variable and is not large enough to contain the result and the NUL-terminator:

- If the *CNULRQD option is specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*YES) on the SET OPTION statement), the following occurs:
  - The result is truncated.
  - The last character is the NUL-terminator.
  - The value 'W' is assigned to SQLWARN1 in the SQLCA.
- If the *NOCNULRQD option on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement) is specified, the following occurs:
  - The NUL-terminator is not returned.
  - The value 'N' is assigned to SQLWARN1 in the SQLCA.

Result column evaluation considerations: If an error occurs while evaluating a result column in the expression list of a VALUES INTO statement as the result of an arithmetic expression (such as division by zero, or overflow) or a numeric or character conversion error, the result is the null value. As in any other case of a null value, an indicator variable must be provided. The value of the variable is undefined. In this case, however, the indicator variable is set to the value of -2. Processing of the statement continues and a warning is returned. If an indicator variable is not provided, an error is returned and no more values are assigned to variables. It is possible that some values have already been assigned to variables and will remain assigned when the error is returned.

119. If assigning to an SQL-variable or SQL-parameter and the standards option is specified, storage assignment rules apply. For information about the standards option, see "Standards compliance" on page xi.

When a datetime value is returned, the length of the variable must be large enough to store the complete value. Otherwise, depending on how much of the value would have to be truncated, a warning or error is returned. See "Datetime assignments" on page 105 for details.

Multiple assignments: If more than one variable is specified in the INTO clause, all *expression*s are evaluated before the assignments are performed. Thus, references to a *variable* in an *expression* are always the value of the *variable* prior to any assignment in the VALUES INTO statement.

Examples

Example 1: Assign the value of the CURRENT PATH special register to host variable HV1.

```
EXEC SQL VALUES CURRENT PATH
         INTO :HV1;
```

Example 2: Assume that LOB locator LOB1 is associated with a CLOB value. Assign a portion of the CLOB value to host variable DETAILS using the LOB locator, and assign CURRENT TIMESTAMP to the host variable TIMETRACK.

```
EXEC SQL VALUES (SUBSTR(:LOB1,1,35), CURRENT TIMESTAMP)
         INTO :DETAILS, :TIMETRACK;
```

WHENEVER

The WHENEVER statement specifies the action to be taken when a specified exception condition occurs.

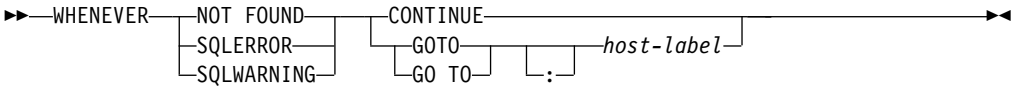
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX. See the Embedded SQL Programming topic collection for information about handling errors in REXX.

Authorization

None required.

Syntax



Description

The NOT FOUND, SQLERROR, or SQLWARNING clause is used to identify the type of exception condition.

NOT FOUND
Identifies any condition that results in an SQLSTATE of '02000' or an SQLCODE of +100.

SQLERROR
Identifies any condition that results in an SQLSTATE value where the first two characters are not '00', '01', or '02'.

SQLWARNING
Identifies any condition that results in an SQLSTATE value where the first two characters are '01', or a warning condition (SQLWARN0 is 'W').

The CONTINUE or GO TO clause are used to specify the next statement to be executed when the identified type of exception condition exists.

CONTINUE
Specifies the next sequential instruction of the source program.

GOTO or GO TO *host-label*
Specifies the statement identified by *host-label*. For host-label, substitute a single token, optionally preceded by a colon. The form of the token depends on the host language. In a COBOL program, for example, it can be a *section-name* or an unqualified *paragraph-name*.

Notes

WHENEVER statement scope: Every executable SQL statement in a program is within the scope of one implicit or explicit WHENEVER statement of each type. The scope of a WHENEVER statement is related to the listing sequence of the statements in the program, not their execution sequence.

An SQL statement is within the scope of the last WHENEVER statement of each type that is specified before that SQL statement in the source program. If a WHENEVER statement of some type is not specified before an SQL statement, that SQL statement is within the scope of an implicit WHENEVER statement of that type in which CONTINUE is specified.

SQL does support nested programs in COBOL, C, and RPG. However, SQL does not honor normal COBOL, C, and RPG scoping rules. That is, the last WHENEVER statement specified in the program source prior to the nested procedure is still in effect for that nested program. The label referenced in the WHENEVER statement must be duplicated within that inner program. Alternatively, the inner program could specify a new WHENEVER statement.

Example

The following statements can be embedded in a COBOL program.

Example 1: Go to the label HANDLER for any statement that produces an error.

```
EXEC SQL  WHENEVER SQLERROR GOTO HANDLER  END-EXEC.
```

Example 2: Continue processing for any statement that produces a warning.

```
EXEC SQL  WHENEVER SQLWARNING CONTINUE  END-EXEC.
```

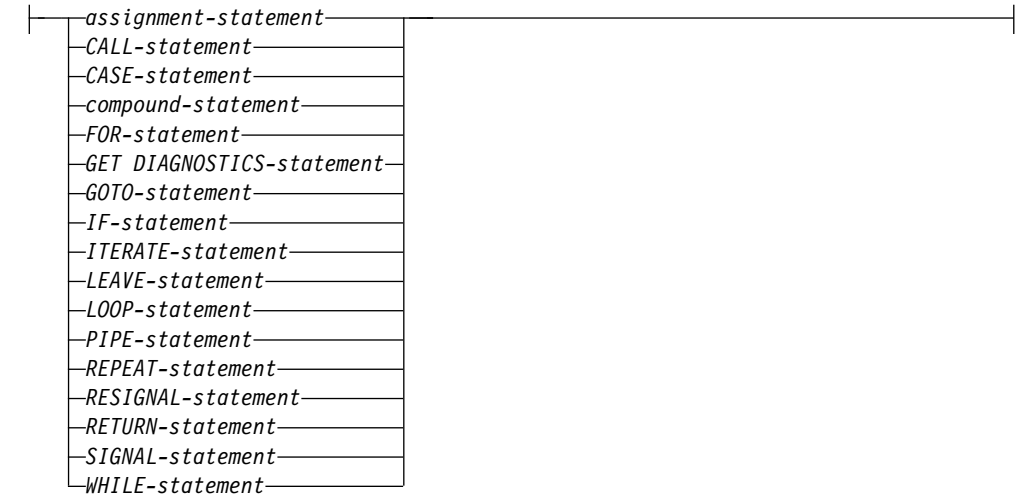
Example 3: Go to the label ENDDATA for any statement that does not return data when expected to do so.

```
EXEC SQL  WHENEVER NOT FOUND GOTO ENDDATA  END-EXEC.
```

Chapter 7. SQL control statements

Control statements are SQL statements that allow SQL to be used in a manner similar to writing a program in a structured programming language. SQL control statements provide the capability to control the logic flow, declare and set variables, and handle warnings and exceptions. Some SQL control statements include other nested SQL statements.

SQL-control-statement:



Control statements are supported in SQL procedures, SQL functions, SQL triggers and compound (dynamic) statements.

SQL procedures are created by specifying an SQL routine body on the CREATE PROCEDURE statement. SQL functions are created by specifying an SQL routine body on the CREATE FUNCTION statement. SQL routines are SQL procedures or SQL functions. SQL triggers are created by specifying an SQL routine body on the CREATE TRIGGER statement. A compound (dynamic) statement is defined by specifying an SQL routine body on the compound (dynamic) statement.

An SQL procedure can also be altered. A new SQL routine body can be specified on the ALTER PROCEDURE statement. An SQL function can also be altered. A new SQL routine body can be specified on the ALTER FUNCTION statement.

An SQL routine body must be a single SQL statement which may be an SQL control statement.

The SQL routine body is the executable part of the procedure, function, or trigger that is transformed by the database manager into a program or service program. When an SQL routine or trigger is created, SQL creates a temporary source file (QTEMP/QSQLSRC and QTEMP/QSQLT00000) that will contain C source code with embedded SQL statements. If either of these source files exist, they will be modified if needed to have the same CCSID as the source. If DBGVIEW(*SOURCE) is specified, SQL creates the root source for the routine or trigger in source file QSQDSRC in the same library as the procedure, function or trigger.

An SQL procedure or SQL trigger is created as a program (*PGM) object using the CRTPGM command. An SQL function is created as a service program (*SRVPGM) object using the CRTSRVPGM command. The program or service program is created in the library that is the implicit or explicit qualifier of the procedure, function, or trigger name.

When the program or service program is created, the SQL statements other than certain control statements become embedded SQL statements in the program or service program. The CALL, SIGNAL, RESIGNAL, and GET DIAGNOSTIC control statements also become embedded SQL statements in the program or service program.

The specified procedure or function is registered in the SYSROUTINES and SYSPARMS catalog tables, and an internal link is created from SYSROUTINES to the program. When the procedure is called using the SQL CALL statement or when the function is invoked in an SQL statement, the program associated with the routine is called. The specified SQL trigger is registered in the SYSTRIGGER catalog table.

References to SQL parameters and SQL variables

SQL parameters and SQL variables can be referenced anywhere in an SQL procedure statement where an expression or variable can be specified.

Host variables cannot be specified in SQL functions, SQL procedures, or SQL triggers. SQL parameters can be referenced anywhere in the routine and can be qualified with the routine name. SQL variables can be referenced anywhere in the compound statement in which they are declared, including any statement that is directly or indirectly nested within that compound statement. If the compound statement where the variable is declared has a label, references to the variable name can be qualified with that label

All SQL parameters and SQL variables are considered nullable except SQL variables that are explicitly declared as NOT NULL. The name of an SQL parameter or SQL variable in an SQL routine can be the same as the name of a column in a table or view referenced in the routine. The name of an SQL variable can also be the same as the name of another SQL variable declared in the same routine. This can occur when the two SQL variables are declared in different *compound-statements*. The *compound-statement* that contains the declaration of an SQL variable determines the scope of that variable. See "compound-statement" on page 1445, for more information.

Names that are the same should be explicitly qualified. Qualifying a name clearly indicates whether the name refers to a column, global variable, SQL variable, or SQL parameter. If the name is not qualified, or qualified but still ambiguous, the following rules describe how the name is resolved. The name is resolved by checking for a match in the following order:

- If the tables and views specified in an SQL routine body exist at the time the routine is created, the name will first be checked as a column name.
- If not found as a column, it will then be checked as an SQL variable name. The SQL variable can be declared within the *compound-statement* that contains the reference, or within a compound statement in which that compound statement is nested. If two SQL variables are within the same scope and have the same name,[120] the SQL variable that is declared in the innermost compound statement is used.
- If not found as an SQL variable name, the name will be checked as an SQL parameter name.

If the name is still not resolved as a column, SQL variable, or SQL parameter and the scope of the name included a table or view that does not exist at the current server, it will be assumed to be a column or global variable. If all the tables and views exist at the current server, it will be assumed to be a global variable. If the SQL_GVAR_BUILD_RULE QAQQINI option is *EXIST and the global variable does not exist, an error will be issued.

The name of an SQL variable or SQL parameter in an SQL routine can be the same as the name of an identifier used in certain SQL statements. If the name is not qualified, the following rules describe whether the name refers to the identifier or to the SQL parameter or SQL variable. Qualified SQL parameters and SQL variables are not supported for these names.

120. Which can happen if they are declared in different compound statements.

- In the SET PATH and SET SCHEMA statements, the name is checked as an SQL parameter name or SQL variable name. If not found as an SQL variable or SQL parameter name, it will then be used as an identifier.
- In the CONNECT, DISCONNECT, RELEASE, and SET CONNECTION statements, the name is used as an identifier.
- In the CALL statement, the name is used as an identifier.
- In the ASSOCIATE LOCATORS and DESCRIBE PROCEDURE statements when used in a CREATE TRIGGER statement, the name is used as an identifier.

References to SQL condition names

The name of an SQL condition can be the same as the name of another SQL condition declared in the same routine. This can occur when the two SQL conditions are declared in different *compound-statements*.

The *compound-statement* that contains the declaration of an SQL condition name determines the scope of that condition name. A condition name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement. A condition name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a condition name, the condition that is declared in the innermost compound statement is the condition that is used. See “compound-statement” on page 1445, for more information.

References to SQL cursor names

The name of an SQL cursor can be the same as the name of another SQL cursor declared in the same routine. This can occur when the two SQL cursors are declared in different *compound-statements*. The cursor name specified in a FOR statement can be the same as the name of another SQL cursor declared in the same *compound-statement*.

The *compound-statement* that contains the declaration of an SQL cursor determines the scope of that cursor name. A cursor name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement. A cursor name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a cursor name, the cursor that is declared in the innermost compound statement is the cursor that is used. See “compound-statement” on page 1445, for more information.

References to SQL labels

Labels can be specified at the beginning of most SQL procedure statements. If a label is specified on an SQL procedure statement, it must be unique from other labels within the same scope. A label must not be the same as any other label within the same compound statement, must not be the same as a label specified on the compound statement itself, and if the compound statement is nested within another compound statement, the label must not be the same as the label specified on any higher level compound statement. The label must not be the same as the name of the SQL function, SQL procedure, or SQL trigger that contains the SQL procedure statement.

Specifying a label for an SQL procedure statement defines that label and determines the scope of that label. A label name can only be referenced within the compound statement in which it is defined, including any statement that is directly or indirectly nested within that compound statement. A label can be specified as the target of a GOTO, LEAVE, or ITERATE statement, subject to the rules for the statement that references the label as a target.

Summary of 'name' scoping in nested compound statements

Nested compound statements can be used within an SQL routine to define the scope of SQL variable declarations, cursors, condition names, and condition handlers.

Additionally, labels have a defined scope in the context of nested compound statements. However the rules for name space, and how non-unique names can be referenced, differs depending on the type of name. The following table summarizes these differences.

Table 110. Summary of 'Name' Scoping in Nested Compound Statements

Type of name	Must be unique within...	Qualification allowed?	Can be referenced within...
SQL variable	the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement.	Yes, can be qualified with the label of the compound statement in which the variable was declared.	the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When multiple SQL variables are defined with the same name you can use a label to explicitly refer to a specific variable that is not the most local in scope.
condition	the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement.	No	the compound statement in which it is declared, including any compound statements that are nested within that compound statement. Can be used in the declaration of a condition handler, or in a SIGNAL or RESIGNAL statement. **Note:** When multiple conditions are defined with the same name there is no way to explicitly refer to the condition that is not the most local in scope.
cursor	the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement.	No	the compound statement in which it is declared, including any compound statements that are nested within that compound statement. **Note:** When multiple cursors are defined with the same name there is no way to explicitly refer to the cursor that is not the most local in scope. However, if the cursor is defined as a result set cursor (for example, the WITH RETURN clause was specified as part of the cursor declaration), the invoking application can access the result set.
label	the compound statement that declared the variable, including any declarations in compound statements that are nested within that compound statement.	No	the compound statement in which it is declared, including any compound statements that are nested within that compound statement. Use a label to qualify the name of an SQL variable or as the target of a GOTO, LEAVE, or ITERATE statement.

SQL-procedure-statement

An SQL control statement may allow multiple SQL statements to be specified within the SQL control statement. These statements are defined as SQL procedure statements.

Syntax

SQL-procedure-statement

Notes:

1. A COMMIT, ROLLBACK, CONNECT, DISCONNECT, SET CONNECTION, or SET RESULT SETS statement is only allowed in an SQL procedure.
2. An ALTER PROCEDURE (SQL), ALTER FUNCTION (SQL Scalar), or ALTER FUNCTION (SQL Table) statement with a REPLACE keyword is not allowed in an *SQL-routine-body*.
3. A SET TRANSACTION statement is only allowed in an SQL function or trigger.
4. A *SET transition-variable-statement* is only allowed in a trigger. A *fullselect* and *VALUES-statement* can also be specified in a trigger.

Notes

Comments: Comments can be included within the body of an SQL procedure. In addition to the double-dash form of comments (--), a comment can begin with /* and end with */. The following rules apply to this form of a comment.

- The beginning characters /* must be adjacent and on the same line.
- The ending characters */ must be adjacent and on the same line.
- Comments can be started wherever a space is valid.
- Comments can be continued to the next line.

Detecting and processing error and warning conditions: As an SQL statement is executed, the database manager stores information about the processing of the statement in a diagnostics area (including the SQLSTATE and SQLCODE), unless otherwise noted in the description of the SQL statement. A completion condition indicates the SQL statement completed successfully, completed with a warning condition, or completed with a not found condition. An exception condition indicates that the SQL statement was not successful.

A condition handler can be defined in a compound statement to execute when an exception condition, a warning condition, or a not found condition occurs. The declaration of a condition handler includes the code that is to be executed when the condition handler is activated. When a condition other than a successful completion occurs in the processing of *SQL-procedure-statement*, if a condition handler that could handle the condition is within scope, one such condition handler will be activated to process the condition. See "compound-statement" on page 1445 for information about defining condition handlers. The code in the condition handler can check for a warning condition, not found condition, or exception condition and take the appropriate action. Use one of the following methods at the beginning of the body of a condition handler to check the condition in the diagnostics area that caused the handler to be activated:

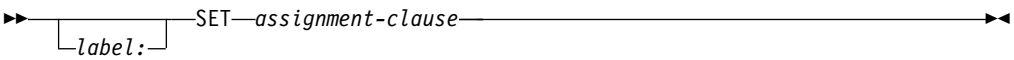
- Issue a GET DIAGNOSTICS statement to request the condition information. See "GET DIAGNOSTICS statement" on page 1457.
- Test the SQL variables SQLSTATE and SQLCODE.

If the condition is a warning and there is not a handler for the condition, the above two methods can also be used outside of the body of a condition handler immediately following the statement for which the condition is wanted. If the condition is an error and there is not a handler for the condition, the routine or trigger terminates with the error condition.

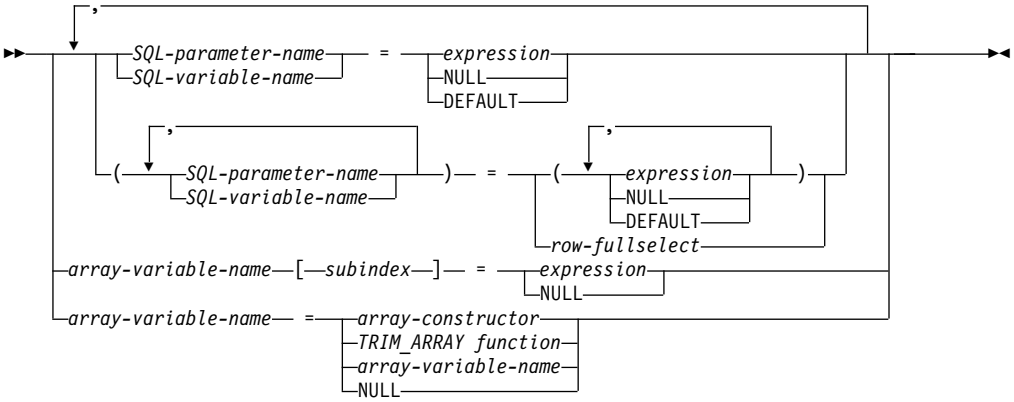
assignment-statement

The *assignment-statement* assigns a value to an SQL parameter or SQL variable.

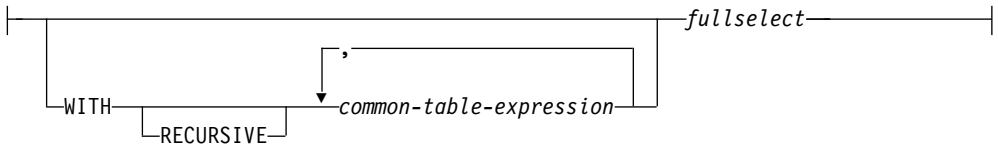
Syntax



assignment-clause:



row-fullselect:



Description

label
Specifies the label for the *assignment-statement* statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

SQL-parameter-name
Identifies the SQL parameter that is the assignment target. The SQL parameter must be specified in *parameter-declaration* in the CREATE PROCEDURE or CREATE FUNCTION statement.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables can only be declared in a *compound-statement* or be a transition variable.

expression **or NULL**
Specifies the expression or value that is the source for the assignment.

DEFAULT
Specifies that the default value for the column associated with the transition variable will be used. This can only be specified for a global variable and in

SQL triggers for transition variables. Only one global variable can be assigned in the SET statement when DEFAULT is used.

row-fullselect
A fullselect that returns a single result row. The result column values are assigned to the corresponding SQL variable or parameter. If the result of the fullselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

WITH *common-table-expression*
Specifies a common table expression. For an explanation of common table expression, see "common-table-expression" on page 683.

fullselect
A *fullselect* that returns a single result row. The result column values are assigned to each corresponding *variable*. If the result of the *fullselect* is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

array-variable-name
Identifies an SQL variable or parameter. The variable or parameter must be of an array type.

[*subindex***]**
Numeric expression that specifies which element in the array will be the target of the assignment. The subindex must be of an exact numeric type with zero scale; it cannot be null. Its value must be between 1 and the maximum cardinality defined for the array.

array-constructor
Specifies the value of an array constructor. See "ARRAY constructor" on page 180.

TRIM_ARRAY function
Specifies the TRIM_ARRAY scalar function. See "TRIM_ARRAY" on page 521.

Notes

Assignment rules: Assignments in the assignment statement must conform to the SQL retrieval assignment rules as described in "Assignments and comparisons" on page 98.[121]

If the assignment is of the form `SET A[idx] = rhs`, where *A* is an array variable name, *idx* is an expression used as the subindex, and *rhs* is an expression of a compatible type as the array element, then:

1. If array *A* is the null value, set *A* to the empty array.
2. Let *C* be the cardinality of array *A*.
3. If *idx* is less than or equal to *C*, the value in the position identified by *idx* is replaced by the value of *rhs*.
4. If *idx* is greater than *C*, then:
   a. The value in position *i*, for *i* greater than *C* and less that *idx*, is set to the null value.
   b. The value in position *idx* is set to the value of *rhs*.
   c. The cardinality of *A* is set to *idx*.

121. If assigning to an SQL-variable or SQL-parameter and the standards option is specified, storage assignment rules apply. For information about the standards option, see "Standards compliance" on page xi.

Assignments involving SQL parameters: An IN parameter can appear on the left or right side of an *assignment-statement*. When control returns to the caller, the original value of the IN parameter is retained. An OUT parameter can also appear on the left or right side of an *assignment-statement*. If used without first being assigned a value, the value is null. When control returns to the caller, the last value that is assigned to an OUT parameter is returned to the caller. For an INOUT parameter, the first value of the parameter is determined by the caller, and the last value that is assigned to the parameter is returned to the caller.

Arrays: If an assignment is to an array or array element, only one assignment is allowed in the statement.

Special Registers: If a variable has been declared with an identifier that matches the name of a special register (such as PATH), then the variable must be delimited to distinguish it from assignment to the special register (for example, SET "PATH" = 1; for a variable called PATH declared as an integer).

Considerations for SQLSTATE and SQLCODE variables: Assignment to these variables is not prohibited; however, it is not recommended as assignment does not affect the diagnostics area or result in the activation of condition handlers. The SQLCODE and SQLSTATE will be reset and the diagnostics area or SQLCA initialized for each *assignment-statement* other than *assignment-statements* that:

- assign the SQLSTATE or SQLCODE variable to another variable or
- set a constant value into the SQLSTATE or SQLCODE variable.

Example

Example 1: Increase the SQL variable p_salary by 10 percent.

```
SET p_salary = p_salary + (p_salary * .10)
```

Example 2: Set SQL variable p_salary to the null value

```
SET p_salary = NULL
```

Example 3: Set the SQL array variable p_phonenumbers to an array of fixed numbers.

```
SET p_phonenumbers = ARRAY[9055553907, 4165554213, 4085553678]
```

Example 4: Set the SQL array variable p_phonenumbers to an array of numbers retrieved from the PHONENUMBER table.

```
SET p_phonenumbers = ARRAY
  [SELECT NUMBER FROM PHONENUMBERS
          WHERE EMPID = 624]
```

*Example 5:*Assign p_mynumber to the first and tenth elements of the SQL array variable p_phonenumbers. After the first assignment, the cardinality of p_phonenumbers is 1. After the second assignment, the cardinality is 10, and elements 2 to 9 have been implicitly assigned the null value.

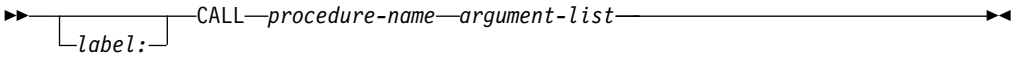
```
SET p_phonenumbers[1] = p_mynumber

 SET p_phonenumbers[10] = p_mynumber
```

CALL statement

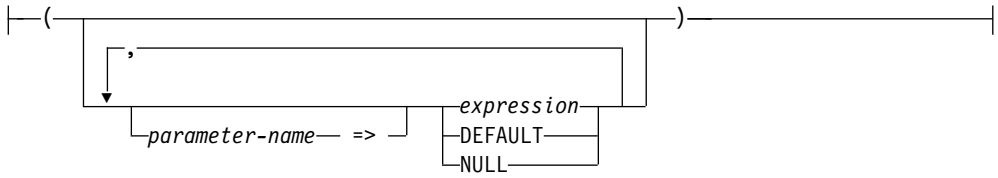
The CALL statement invokes a procedure. The syntax of CALL in an SQL function, SQL procedure, or SQL trigger is a subset of what is supported as a CALL statement in other contexts.

See "CALL" on page 803 for details.

Syntax



argument-list:



Description

label
Specifies the label for the CALL statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

procedure-name
Identifies the procedure to call. The *procedure-name* must identify a procedure that exists at the current server.

argument-list
Identifies a list of values to be passed as parameters to the procedure. The *n*th unnamed argument corresponds to the *n*th parameter in the procedure.

Each parameter defined (using CREATE PROCEDURE) as OUT must be specified as either a *SQL-variable-name* or a *SQL-parameter-name*. A default cannot be specified for an OUT parameter. If a default is used for an INOUT parameter, then the default expression is used to initialize the parameter for input to the procedure. No value is returned for this parameter when the procedure exits.

When a procedure is called, arguments must be specified for all parameters that are not defined to have a default value. When an argument is assigned to a parameter using the named syntax, then all the arguments that follow it must also be assigned using the named syntax.

Any references to date, time, or timestamp special register values in the argument list will use one clock reading for any default expressions and a separate clock reading for any references in the explicit arguments.

The application requester assumes all parameters that are variables are INOUT parameters. All parameters that are not variables are assumed to be input parameters. The actual attributes of the parameters are determined by the current server.

parameter-name
Name of the parameter to which the argument value is assigned. The name must match a parameter name defined for the procedure. Named arguments correspond to the same named parameter regardless of the order in which they are specified in the argument list. When an argument is assigned to a parameter by name, all the arguments that follow it must also be assigned by name.

A named argument must be specified only one time (implicitly or explicitly).

Named arguments are not allowed on a call to a procedure that was not defined using a CREATE PROCEDURE statement.

expression
An *expression* of the type described in "Expressions" on page 165, that does not include an aggregate function or column name.

DEFAULT
Specifies the default as defined in the CREATE PROCEDURE statement is used as an argument to the procedure. If no default is defined for the parameter, the NULL value is used.

NULL
Specifies a null value as an argument to the procedure.

Notes

Rules for arguments to OUT and INOUT parameters: Each OUT or INOUT parameter must be specified as an SQL parameter or SQL variable.

Special registers: The initial value of a special register in a procedure is inherited from the caller of the procedure. A value assigned to a special register within the procedure is used for the entire SQL procedure and will be inherited by any procedure subsequently called from that procedure. When a procedure returns to its caller, the special registers are restored to the original values of the caller.

Related information: See "CALL" on page 803 for more information.

Example

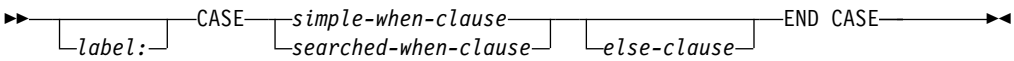
Call procedure *proc1* and pass SQL variables as parameters.

```
CALL proc1(v_empno, v_salary)
```

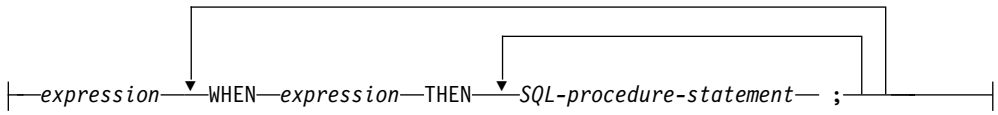
CASE statement

The CASE statement selects an execution path based on multiple conditions.

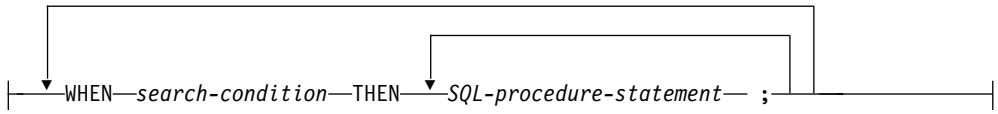
Syntax



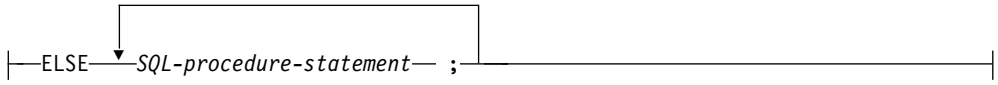
simple-when-clause:



searched-when-clause:



else-clause:



Description

label
Specifies the label for the CASE statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

simple-when-clause
The value of the *expression* prior to the first WHEN keyword is tested for equality with the value of each *expression* that follows the WHEN keyword. If the comparison is true, the statements in the associated THEN clause are executed and processing of the CASE statement ends. If the result is unknown or false, processing continues to the next comparison. If the result does not match any of the comparisons, and an ELSE clause is present, the statements in the ELSE clause are executed.

searched-when-clause
The *search-condition* following the WHEN keyword is evaluated. If it evaluates to true, the statements in the associated THEN clause are executed and processing of the CASE statement ends. If it evaluates to false, or unknown, the next *search-condition* is evaluated. If no *search-condition* evaluates to true and an ELSE clause is present, the statements in the ELSE clause are executed.

else-clause
If none of the conditions specified in the *simple-when-clause* or *searched-when-clause* are true, then the statements in the *else-clause* are executed.

If none of the conditions specified in the WHEN are true, and an ELSE clause is not specified, an error is issued at runtime, and the execution of the CASE statement is terminated (SQLSTATE 20000).

SQL-procedure-statement
Specifies a statement to execute. See "SQL-procedure-statement" on page 1435.

Notes

Nesting of CASE statements: CASE statements that use a *simple-when-clause* can be nested up to three levels. CASE statements that use a *searched-when-clause* have no limit to the number of nesting levels.

Considerations for SQLSTATE and SQLCODE variables: When the first *SQL-procedure-statement* in the CASE statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the *expressions* or *search-conditions* of that CASE statement. If a CASE statement does not include an ELSE clause and none of the *search-conditions* evaluate to true, then any error returned from the *expression* is returned.

Examples

Example 1: Depending on the value of SQL variable v_workdept, update column DEPTNAME in table DEPARTMENT with the appropriate name.

The following example shows how to do this using the syntax for a *simple-when-clause*.

```
CASE v_workdept
  WHEN 'A00'
    THEN UPDATE department SET
                deptname = 'DATA ACCESS 1';
  WHEN 'B01'
    THEN UPDATE department SET
                deptname = 'DATA ACCESS 2';
    ELSE UPDATE department SET
                deptname = 'DATA ACCESS 3';
END CASE
```

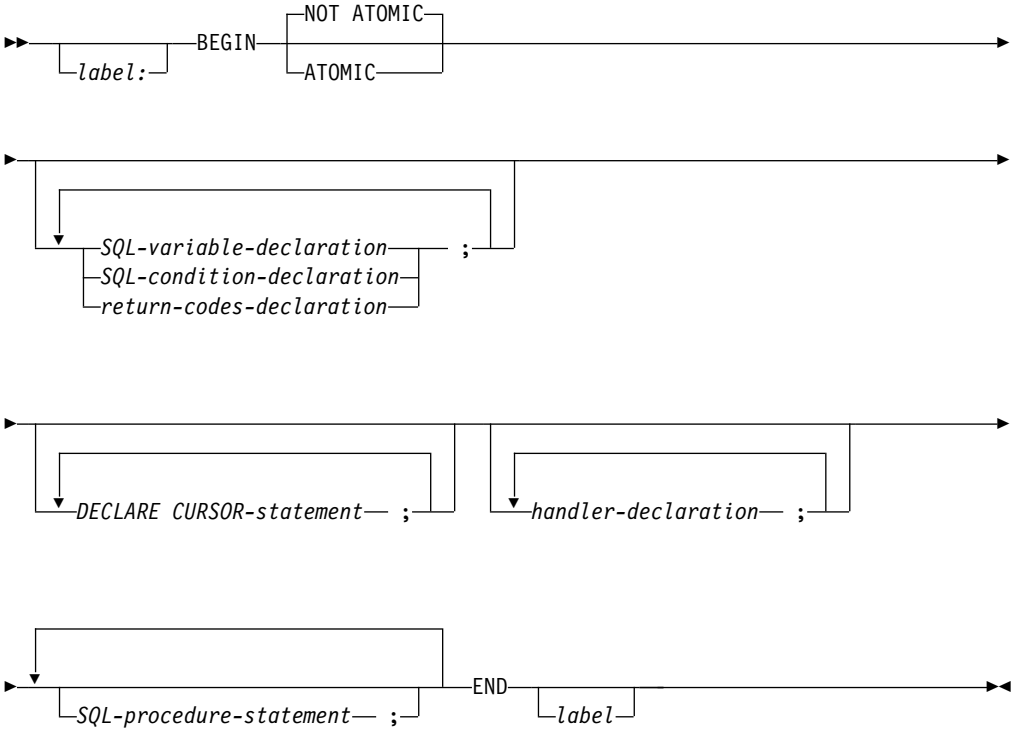
Example 2: The following example shows how to do this using the syntax for a *searched-when-clause*:

```
CASE
  WHEN v_workdept = 'A00'
    THEN UPDATE department SET
                deptname = 'DATA ACCESS 1';
  WHEN v_workdept = 'B01'
    THEN UPDATE department SET
                deptname = 'DATA ACCESS 2';
    ELSE UPDATE department SET
                deptname = 'DATA ACCESS 3';
END CASE
```

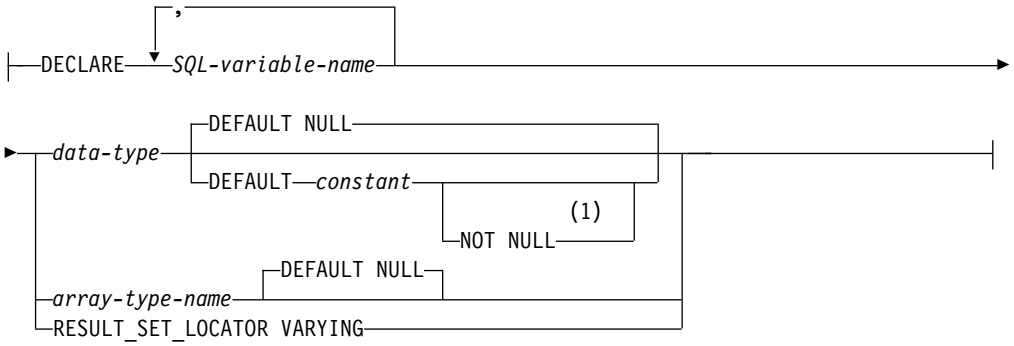
compound-statement

A compound statement groups other statements together in an SQL procedure. A compound statement allows the declaration of SQL variables, cursors, and condition handlers.

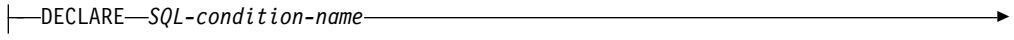
Syntax

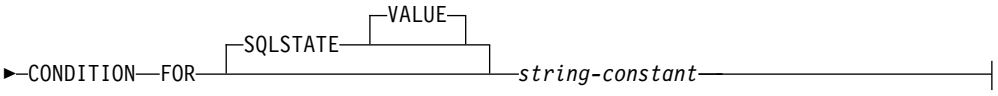


SQL-variable-declaration:

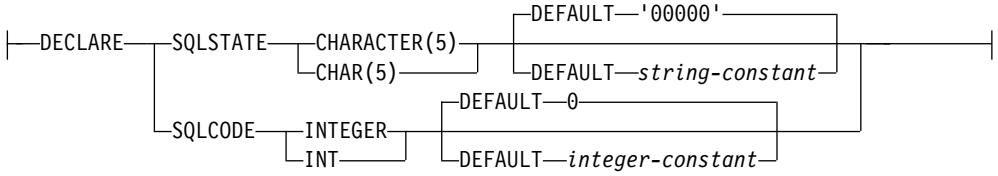


SQL-condition-declaration:

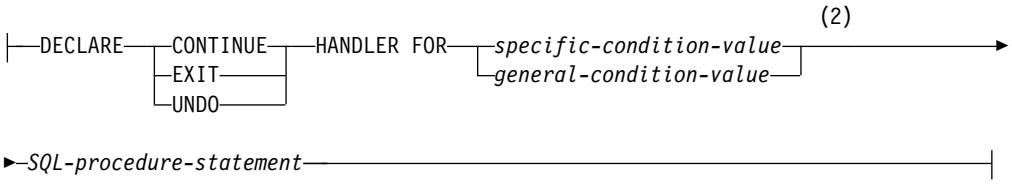




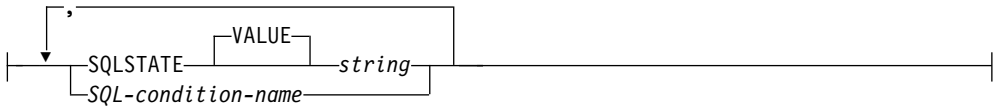
return-codes-declaration:



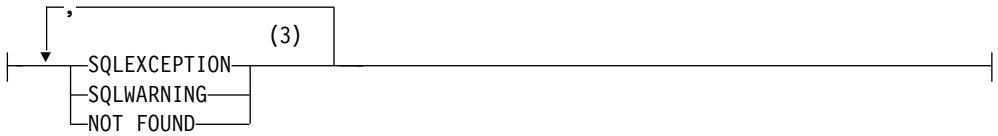
handler-declaration:



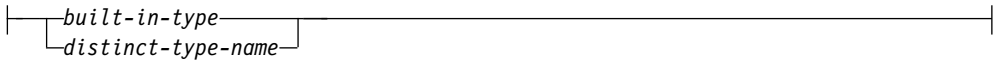
specific-condition-value:



general-condition-value:



data-type:



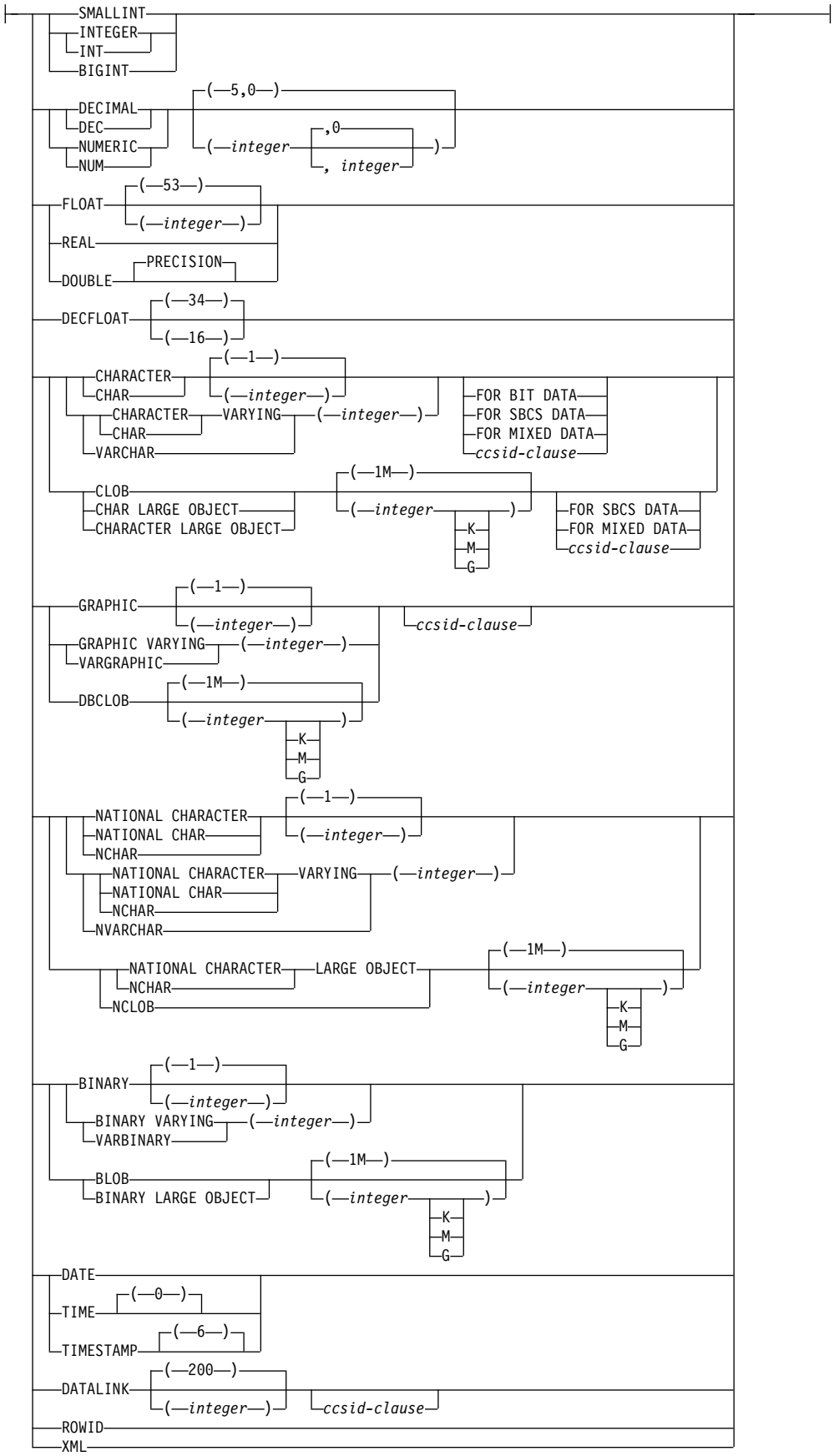
Notes:

1 The DEFAULT and NOT NULL clauses can be specified in either order.

2 *specific-condition-value* and *general-condition-value* cannot be specified in the same handler declaration.

3 The same clause must not be specified more than once.

built-in-type:



ccsid-clause:

Description

label
Specifies the label for the *compound-statement*. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

ATOMIC
ATOMIC indicates that an unhandled exception condition within the *compound-statement* causes the *compound-statement* to be rolled back. If ATOMIC is specified, COMMIT or ROLLBACK statements cannot be specified in the compound statement (ROLLBACK TO SAVEPOINT may be specified).

If ATOMIC is specified in an SQL procedure or function, the routine must have a data classification of MODIFIES SQL DATA.

NOT ATOMIC
NOT ATOMIC indicates that an unhandled exception condition within the *compound-statement* does not cause the *compound-statement* to be rolled back. If NOT ATOMIC is specified in the outermost compound statement of an SQL trigger, it is treated as ATOMIC.

SQL-variable-declaration
Declares a variable that is local to the *compound-statement*.

SQL-variable-name
Defines the name of a local SQL variable. The database manager converts all undelimited SQL variable names to uppercase. The name must not be the same as another SQL variable within the same *compound-statement*, excluding any declarations in *compound-statements* nested within the *compound-statement*. Do not name SQL variables the same as column names or parameter names. See "References to SQL parameters and SQL variables" on page 1429 for how SQL variable names are resolved when there are columns with the same name involved in a statement. Variable names should not begin with 'SQL'.

data-type
Specifies the data type of the variable. See "CREATE TABLE" on page 982 for a description of data type.

If the *data-type* is a graphic string data type, consider specifying CCSID 1200 or 13488 to indicate UTF-16 or UCS-2 data. If a CCSID is not specified, the CCSID of the graphic string variable will be the associated DBCS CCSID for the job.

array-type-name
Specifies that the SQL variable is an array as defined with the CREATE TYPE (Array) statement.

DEFAULT *constant* **or NULL**
Defines the default for the SQL variable. The specified constant must represent a value that could be assigned to the variable in accordance with the rules of assignment as described in "Assignments and comparisons" on page 98

page 98. The variable is initialized when the *compound-statement* in which it is declared is entered. If a default value is not specified, the SQL variable is initialized to NULL. SQL variables of type XML cannot have a default value specified. SQL variables of *array-type* are always initialized to NULL.

NOT NULL
Prevents the SQL variable from containing the NULL value. Omission of NOT NULL implies that the variable can be null. SQL variables of type XML cannot have NOT NULL specified

RESULT_SET_LOCATOR VARYING
Specifies the data type for a result set locator variable.

SQL-condition-declaration
Declares a condition name and corresponding SQLSTATE value.

SQL-condition-name
Specifies the name of the condition. The condition name must be unique within the *compound-statement* (excluding any declarations in *compound-statements* nested within the *compound-statement*) in which it is declared.

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE associated with this condition. The string constant must be specified as 5 characters. The SQLSTATE class (the first 2 characters) must not be '00'.

return-codes-declaration
Declares special SQL variables called SQLSTATE and SQLCODE that are set for the first condition in the diagnostics area after executing an SQL statement other than GET DIAGNOSTICS or an empty *compound-statement*. The SQLSTATE and SQLCODE variables can only be declared in the outermost *compound-statement* of an SQL procedure, SQL function, or SQL trigger.

The SQLSTATE and SQLCODE special variables are only intended to be used as a means of obtaining the SQL return codes that resulted from processing the previous SQL statement other than GET DIAGNOSTICS. If there is any intention to use the SQLSTATE and SQLCODE values, save the values immediately to other SQL variables to avoid having the values replaced by the SQL return codes returned after executing the next SQL statement. If a handler is defined that handles an SQLSTATE, you can use an assignment statement to save that SQLSTATE (or the associated SQLCODE) value in another SQL variable, if the assignment is the first statement in the handler.

Assignment to these variables is not prohibited; however, it is not recommended. Assignment to these special variables is ignored by condition handlers. The SQLSTATE and SQLCODE special variables cannot be set to NULL.

DECLARE CURSOR-statement
Declares a cursor in the routine body. The cursor name must be unique within the *compound-statement*, excluding any declarations in *compound-statements* nested within the *compound-statement*.

A *cursor-name* can only be referenced within the *compound-statement* in which it is declared, including any *compound-statements* nested within the *compound-statement*.

Use an OPEN statement to open the cursor, and a FETCH statement to read rows using the cursor. If the cursor in an SQL procedure and is intended for use as a result set:

- specify WITH RETURN when declaring the cursor
- create the procedure using the DYNAMIC RESULT SETS clause with a non-zero value
- do not specify a CLOSE statement in the *compound-statement*.

Any open cursor that does not meet these criteria is closed at the end of the *compound-statement*.

For more information about declaring a cursor, refer to "DECLARE CURSOR" on page 1079.

handler-declaration
Specifies a *handler*, an *SQL-procedure-statement* to execute when an exception or completion condition occurs in the *compound-statement*.

A condition handler declaration cannot reference the same condition value or SQLSTATE value more than once, and cannot reference an SQLSTATE value and a condition name that represent the same SQLSTATE value. For a list of SQLSTATE values as well as more information, see the SQL messages and codes topic collection.

Furthermore, when two or more condition handlers are declared in a compound statement, no two condition handler declarations may specify the same:

- general condition category or
- specific condition, either as an SQLSTATE value or as a condition name that represents the same value.

A condition handler is active for the set of *SQL-procedure-statements* that follow the *handler-declarations* within the *compound-statement* in which it is declared, including any nested compound statements.

A handler for a condition may exist at several levels of nested compound statements. For example, assume that compound statement *n1* contains another compound statement *n2* which contains another compound statement *n3*. When an exception condition occurs within *n3*, any active handlers within *n3* are first allowed to handle the condition. If no appropriate handler exists in *n3*, then the condition is resignalled to *n2* and the active handlers within *n2* may handle the condition. If no appropriate handler exists in *n2*, then the condition is resignalled to *n1* and the active handlers within *n1* may handle the condition. If no appropriate handler exists in *n1*, the condition is considered unhandled.

There are three types of condition handlers:

CONTINUE
Specifies that after the condition handler is activated and completes successfully, control is returned to the SQL statement following the one that raised the exception. If the error occurs while executing a comparison as in an IF, CASE, FOR, WHILE, or REPEAT, control returns to the statement following the corresponding END IF, END CASE, END FOR, END WHILE, or END REPEAT.

EXIT
Specifies that after the condition handler is activated and completes successfully, control is returned to the end of the compound statement that declared the condition handler.

UNDO
Specifies that when the condition handler is activated changes made by the

compound-statement are rolled back. When the handler completes successfully, control is returned to the end of the *compound-statement*. If UNDO is specified, then ATOMIC must be specified.

UNDO cannot be specified in the outermost *compound-statement* of an SQL function or SQL trigger.

The conditions under which the handler is activated are:

SQLSTATE *string*
Specifies that the handler is invoked when the specific SQLSTATE occurs. The SQLSTATE class (the first 2 characters) must not be '00'.

SQL-condition-name
Specifies that the handler is invoked when the specific SQLSTATE associated with the condition name occurs. The *SQL-condition-name* must be previously defined in a *SQL-condition-declaration*.

SQLEXCEPTION
Specifies that the handler is invoked when an exception condition occurs. An exception condition is represented by an SQLSTATE value where the first two characters are not '00', '01', or '02'.

SQLWARNING
Specifies that the handler is invoked when a warning condition occurs. A warning condition is represented by an SQLSTATE value where the first two characters are '01'.

NOT FOUND
Specifies that the handler is invoked when a NOT FOUND condition occurs. A NOT FOUND condition is represented by an SQLSTATE value where the first two characters are '02'.

Notes

Nesting compound statements: Compound statements can be nested. Nested compound statements can be used to scope variable definitions, condition names, condition handlers, and cursors to a subset of the statements in the *compound-statement*. This can simplify the processing done for each SQL procedure statement. Support for nested compound statements enables the use of a compound statement within the declaration of a condition handler.

Condition handlers: Condition handlers in a *compound-statement* are similar to WHENEVER statements used in external SQL application programs. A condition handler can be defined to automatically get control when an exception, warning, or not found condition occurs. The body of a condition handler contains code that is executed when the condition handler is activated. A condition handler can be activated as a result of an exception, warning, or not found condition that is returned by the database manager for the processing of an SQL statement, or the activating condition can be the result of a SIGNAL or RESIGNAL statement issued within the procedure body.

A condition handler is declared within a compound statement, and it is active for the set of *SQL-procedure-statements* that follow all of the condition handler declarations within the compound statement in which the condition handler is declared. To be more specific, the scope of a condition handler declaration H is the list of *SQL-procedure-statements* that follows the condition handler declarations contained within the compound statement in which H appears. This means that the scope of H does not include the statements contained in the body of the

condition handler H, implying that a condition handler cannot handle conditions that arise inside its own body. Similarly, for any two condition handlers H1 and H2 declared in the same compound statement, H1 will not handle conditions arising in the body of H2, and H2 will not handle conditions arising in the body of H1.

The declaration of a condition handler specifies the condition that activates it, the type of the condition handler (CONTINUE, EXIT, or UNDO), and the handler action. The type of the condition handler determines where control is returned to after successful completion of the handler action.

Condition handler activation: When a condition other than successful completion occurs in the processing of an *SQL-procedure-statement*, if a condition handler that could handle the condition is within scope, one such condition handler will be activated to process the condition.

In a routine with nested compound statements, condition handlers that could handle a specific condition may exist at several levels of the nested compound statements. The condition handler that is activated is a condition handler that is declared innermost to the scope in which the condition was encountered. If more than one condition handler at that nesting level could handle the condition, the condition handler that is activated is the most appropriate handler declared in that compound statement.

The most appropriate handler is a handler that is defined in the *compound-statement* which most closely matches the SQLSTATE of the exception or completion condition.

For example, if the innermost compound statement declares a specific handler for SQLSTATE 22001 as well as a handler for SQLEXCEPTION, the specific handler for SQLSTATE 22001 is the most appropriate handler when an SQLSTATE 22001 is encountered. In this case, the specific handler is activated.

When a condition handler is activated, the condition handler action is executed. If the handler action completes successfully or with an unhandled warning, the diagnostics area is cleared, and the type of the condition handler (CONTINUE, EXIT, or UNDO handler) determines where control is returned. Additionally, the SQLSTATE and SQLCODE SQL variables are cleared when a handler completes successfully or with an unhandled warning.

If the handler action does not complete successfully, and an appropriate handler exists for the condition encountered in the handler action, that condition handler is activated. Otherwise, the condition encountered within the condition handler is unhandled.

Unhandled conditions: If a condition is encountered and an appropriate handler does not exist for that condition, the condition is unhandled.

- If the unhandled condition is an exception, the SQL procedure, SQL function, or SQL trigger containing the failing statement is terminated with an unhandled exception condition.
- If the unhandled condition is a warning or not found condition, processing continues with the next statement. Note that the processing of the next SQL statement will cause information about the unhandled condition in the diagnostics area to be overwritten, and evidence of the unhandled condition will no longer exist.

Considerations for using SIGNAL or RESIGNAL statements with nested compound statements: If an *SQL-procedure-statement* specified in the condition handler is either a SIGNAL or RESIGNAL statement with an exception SQLSTATE, the compound statement terminates with the specified exception. This happens even if this condition handler or another condition handler in the same compound statement specifies CONTINUE, since these condition handlers are not in the scope of this exception. If the compound statement is nested in another compound statement, condition handlers in the higher level compound statement may handle the exception because those condition handlers are within the scope of the exception.

Null values in SQL parameters and SQL variables: If the value of an SQL parameter or SQL variable is null and it is used in an SQL statement (such as CONNECT or DESCRIBE) that does not allow an indicator variable, an error is returned.

Effect on open cursors: Upon exit from the *compound-statement* for any reason, all open cursors that are declared in that compound statement are closed, unless they are declared to return result sets or unless *ENDACTGRP is specified.

Considerations for SQLSTATE and SQLCODE SQL variables: The compound statement itself does not affect the SQLSTATE and SQLCODE SQL variables. However, SQL statements contained within the compound statement can affect the SQLSTATE and SQLCODE SQL variables. At the end of the compound statement the SQLSTATE and SQLCODE SQL variables reflect the result of the last SQL statement executed within that compound statement that caused a change to the SQLSTATE and SQLCODE SQL variables. If the SQLSTATE and SQLCODE variables were not changed within the compound statement, they contain the same values as when the compound statement was entered.

Examples

Create a procedure body with a compound statement that performs the following actions.

1. Declares SQL variables.
2. Declares a cursor to return the salary of employees in a department determined by an IN parameter.
3. Declares an EXIT handler for the condition NOT FOUND (end of file) which assigns the value 6666 to the OUT parameter `medianSalary`.
4. Select the number of employees in the given department into the SQL variable `v_numRecords.`
5. Fetch rows from the cursor in a WHILE loop until 50% + 1 of the employees have been retrieved.
6. Return the median salary.

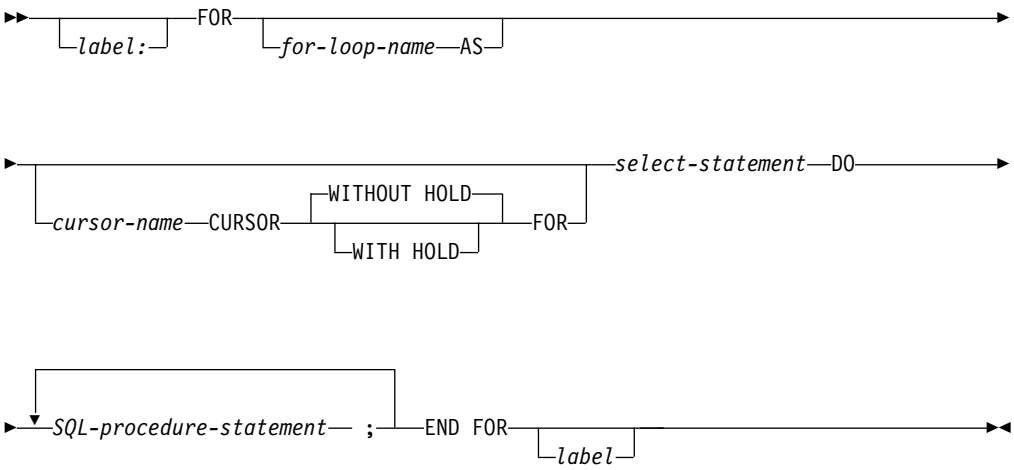
```
CREATE PROCEDURE DEPT_MEDIAN
   (IN  deptNumber   SMALLINT,
    OUT medianSalary DOUBLE)
   LANGUAGE SQL
   BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE c1 CURSOR FOR
       SELECT salary FROM staff
          WHERE DEPT = deptNumber
          ORDER BY salary;
    DECLARE EXIT HANDLER FOR NOT FOUND
```

```
   SET medianSalary = 6666;
   /* initialize OUT parameter */
   SET medianSalary = 0;
   SELECT COUNT(*) INTO v_numRecords FROM staff
      WHERE DEPT = deptNumber;
   OPEN c1;
   WHILE v_counter < (v_numRecords / 2 + 1) DO
      FETCH c1 INTO medianSalary;
      SET v_counter = v_counter + 1;
   END WHILE;
   CLOSE c1;
END
```

FOR statement

The FOR statement executes a statement or group of statements for each row of a table.

Syntax



Description

label
Specifies the label for the FOR statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

for-loop-name
Specifies the label for the implicit *compound-statement* that is generated to implement the FOR statement. It follows the rules for the label of a *compound-statement* except that it cannot be used with an ITERATE, GOTO, or LEAVE statement within the FOR statement. The *for-loop-name* is used to qualify the column names returned by the specified *select-statement*. It must not be the same as any label within the same scope. For more information, see "References to SQL labels" on page 1433.

Either the *for-loop-name* or *label* can be used to qualify other SQL variable names in the statement.

If *for-loop-name* is specified, then it should be used to qualify any other SQL variable names in the statement when debugging the SQL function, SQL procedure, or SQL trigger.

cursor-name
Names a cursor.

WITH HOLD
Prevents the cursor from being closed as a consequence of a commit operation. A cursor declared using the WITH HOLD clause is implicitly closed at commit time only if the connection associated with the cursor is ended during the commit operation. For more information, see "DECLARE CURSOR" on page 1079.

select-statement
Specifies the select statement of the cursor.

Each expression in the select list must have a name. If an expression is not a simple column name, the AS clause must be used to name the expression. If the AS clause is specified, that name is used for the variable and must be unique.

SQL-procedure-statement
Specifies the SQL statements to be executed for each row of the result table of the cursor. The SQL statements should not include an OPEN, FETCH, or CLOSE specifying the cursor name of the FOR statement.

Notes

FOR statement rules: The FOR statement executes one or multiple statements for each row in a result table of the cursor. The cursor is defined by specifying a select list that describes the columns and rows selected. The statements within the FOR statement are executed for each row selected.

The select list must consist of unique column names and the objects referenced in the *select-statement* must exist when the function, procedure, or trigger is created.

The cursor specified in a FOR statement cannot be referenced outside the FOR statement and cannot be specified on an OPEN, FETCH, or CLOSE statement.

Handler warning: Handlers may be used to handle errors that might occur on the open of the cursor or fetch of a row using the cursor in the FOR statement. Handlers defined to handle these open or fetch conditions should not be CONTINUE handlers as they may cause the FOR statement to loop indefinitely.

Example

In this example, the FOR statement is used to specify a cursor that selects 3 columns from the `employee` table. For every row selected, SQL variable *fullname* is set to the last name followed by a comma, the first name, a blank, and the middle initial. Each value for *fullname* is inserted into table `TNAMES`.

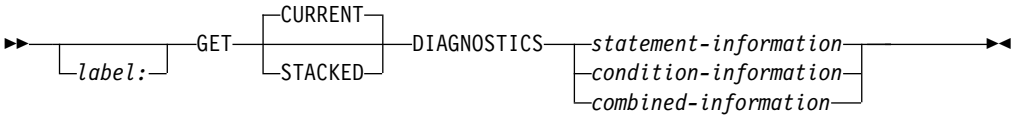
```
BEGIN
  DECLARE fullname CHAR(40);
  FOR v1 AS
      c1 CURSOR FOR
    SELECT firstnme, midinit, lastname FROM employee
     DO
       SET fullname =
          lastname || ', ' || firstnme ||' ' || midinit;
       INSERT INTO TNAMES VALUES ( fullname );
     END FOR;
END;
```

GET DIAGNOSTICS statement

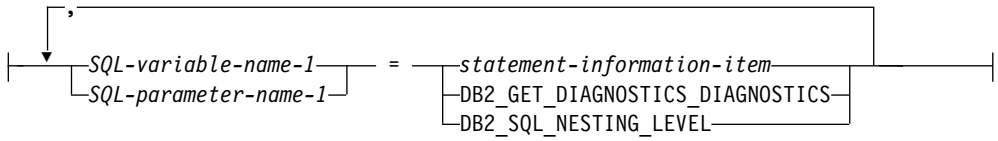
The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed. The syntax of GET DIAGNOSTICS in an SQL function, SQL procedure, or SQL trigger is a subset of what is supported as a GET DIAGNOSTICS statement in other contexts.

See "GET DIAGNOSTICS" on page 1194 for details.

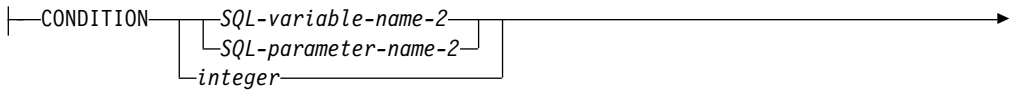
Syntax

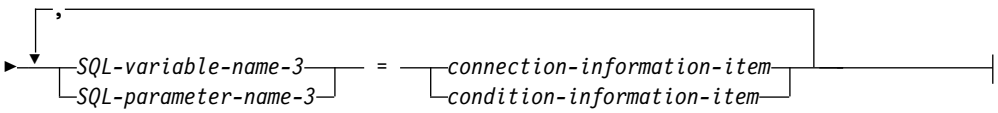


statement-information:

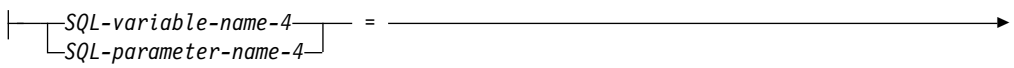


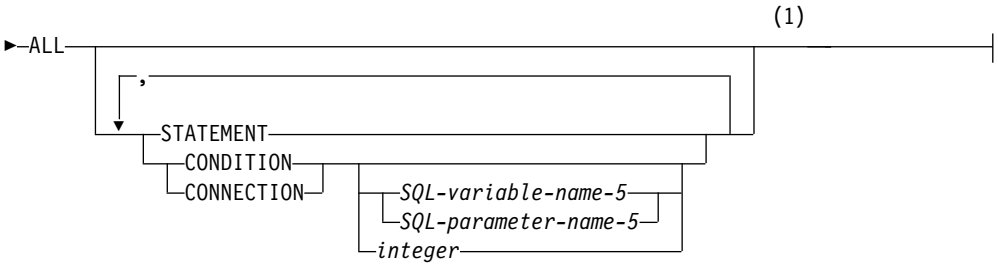
condition-information:





combined-information:



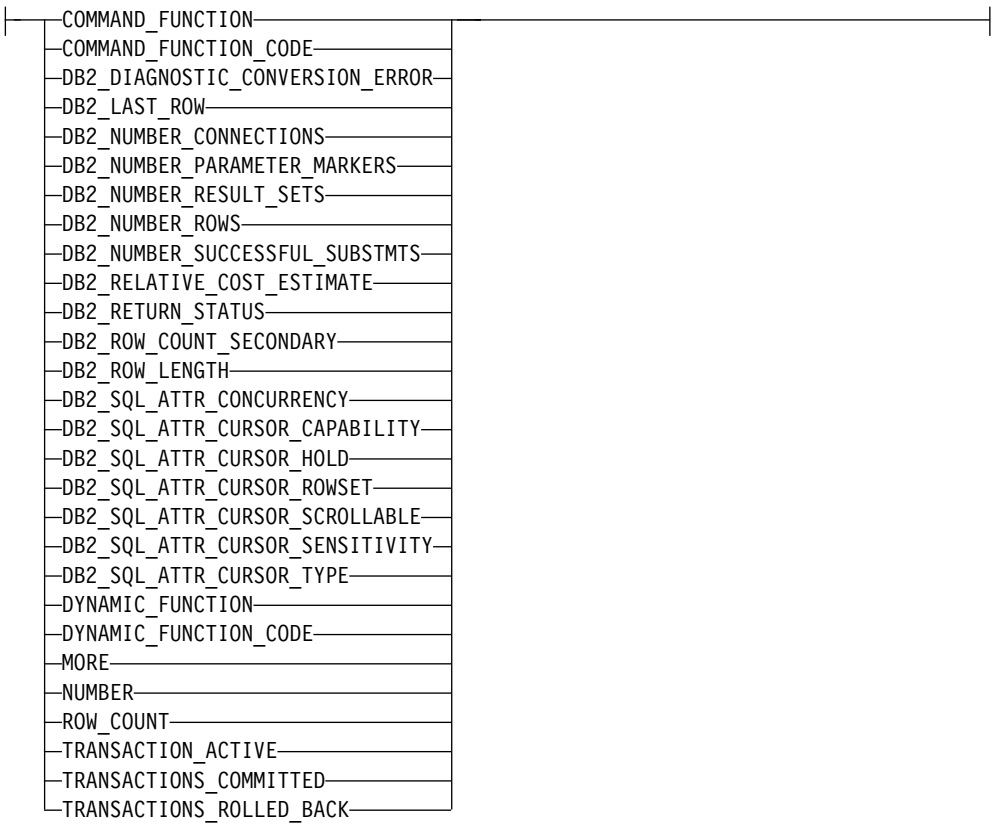


Notes:

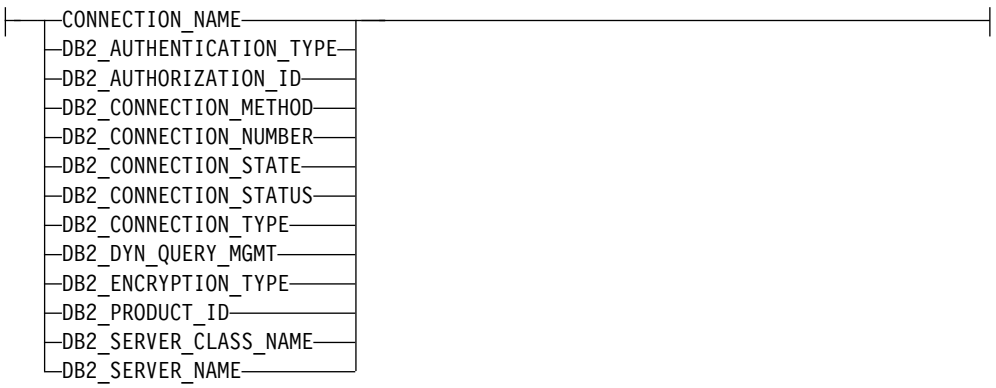
1 STATEMENT can only be specified once. If *SQL-variable-name-5*,

SQL-parameter-name-5, or *integer* is not specified, CONDITION and CONNECTION can only be specified once.

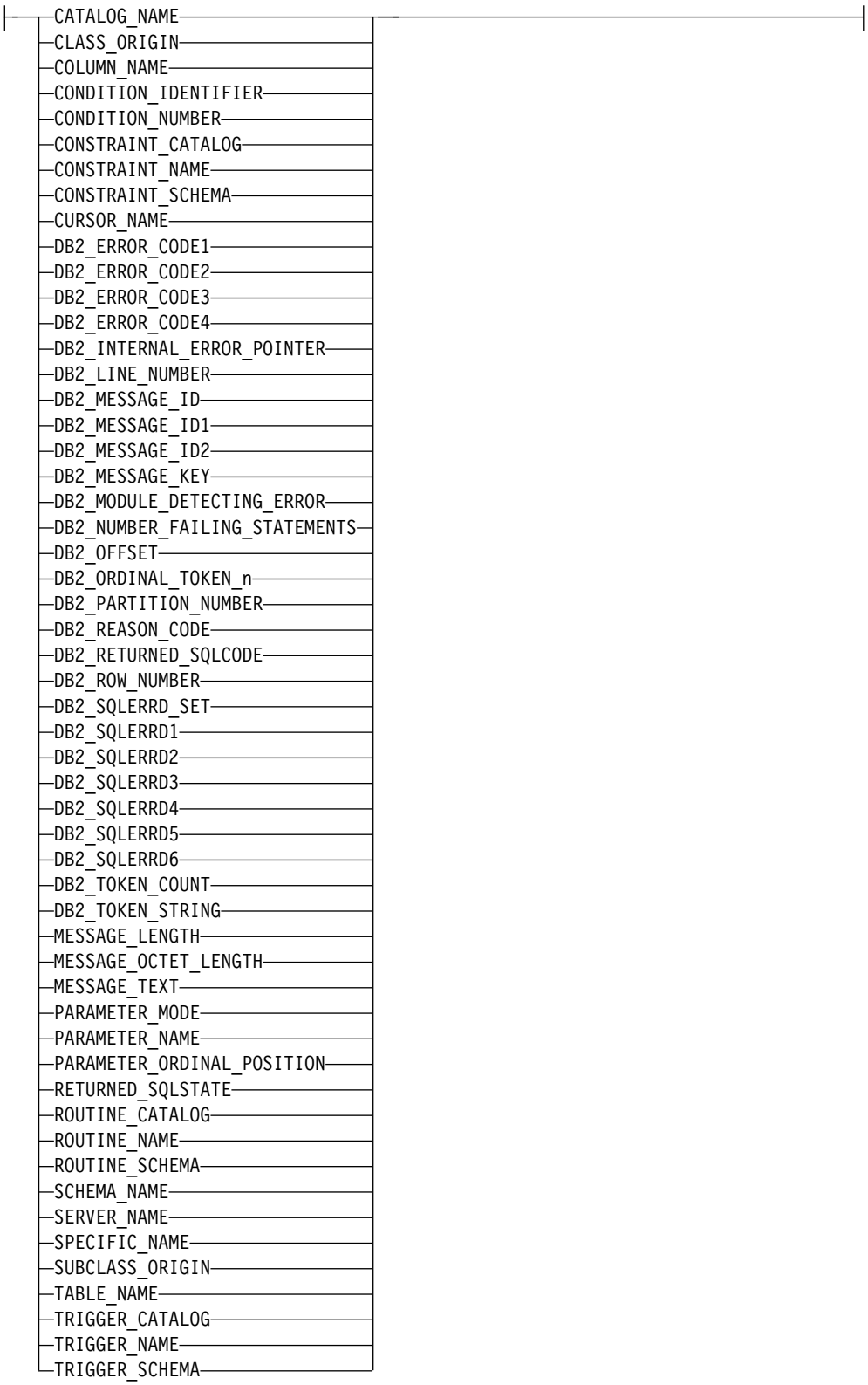
statement-information-item:



connection-information-item:



condition-information-item:



Description

label

Specifies the label for the GET DIAGNOSTICS statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be

the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

CURRENT or STACKED
Specifies which diagnostics area to access.

CURRENT
Specifies to access the first diagnostics area. It corresponds to the previous SQL statement that was executed and that was not a GET DIAGNOSTICS statement. This is the default.

STACKED
Specifies to access the second diagnostics area. The second diagnostics area is only available within a handler. It corresponds to the previous SQL statement that was executed before the handler was entered and that was not a GET DIAGNOSTICS statement. If the GET DIAGNOSTICS statement is the first statement within a handler, then the first diagnostics area and the second diagnostics area contain the same diagnostics information.

statement-information
Returns information about the last SQL statement executed.

SQL-variable-name-1 **or** *SQL-parameter-name-1*
Identifies a variable described in the program in accordance with the rules for declaring SQL variables and SQL parameters. The data type of the SQL variable or SQL parameter must be compatible with the data type as specified in Table 85 on page 1211 for the specified condition information item. The variable is assigned the value of the specified statement information item according to the retrieval assignment rules described in "Retrieval assignment" on page 102. If the value is truncated when assigning it to the SQL variable or SQL parameter, a warning is returned (SQLSTATE 01004) and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition.

If a specified diagnostic item does not contain diagnostic information, then the SQL variable or SQL parameter is set to a default value, based on its data type: 0 for an exact numeric diagnostic item, an empty string for a VARCHAR diagnostic item and blanks for a CHAR diagnostic item.

condition-information
Returns information about the condition or conditions that occurred when the last SQL statement was executed.

CONDITION *SQL-variable-name-2* **or** *SQL-parameter-name-2* **or** *integer*
Identifies the diagnostic for which information is requested. Each diagnostic that occurs while executing an SQL statement is assigned an integer. The value 1 indicates the first diagnostic, 2 indicates the second diagnostic and so on. If the value is 1, then the diagnostic information retrieved corresponds to the condition indicated by the SQLSTATE value actually returned by the execution of the previous SQL statement (other than a GET DIAGNOSTICS statement). The SQL variable or SQL parameter specified must be described in the program in accordance with the rules for declaring exact numeric variables with zero scale. SQL variables or SQL parameters. The value specified must not be less than one or greater than the number of available diagnostics

SQL-variable-name-3 **or** *SQL-parameter-name-3*
Identifies a variable described in the program in accordance with the rules for declaring SQL variables or SQL parameters. The data type of the SQL variable or SQL parameter must be compatible with the data type as

specified in Table 85 on page 1211 for the specified condition information item. The SQL variable or SQL parameter is assigned the value of the specified condition information item according to the retrieval assignment rules described in "Retrieval assignment" on page 102. If the value is truncated when assigning it to the SQL variable or SQL parameter, a warning is returned (SQLSTATE 01004) and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition.

If a specified diagnostic item does not contain diagnostic information, then the SQL variable or SQL parameter is set to a default value, based on its data type: 0 for an exact numeric diagnostic item, an empty string for a VARCHAR diagnostic item and blanks for a CHAR diagnostic item.

combined-information
Returns multiple information items combined into one string.

If the GET DIAGNOSTICS statement is specified in an SQL function, SQL procedure, or trigger, the GET DIAGNOSTICS statement must be the first statement specified in the handler that will handle the error.

If information is wanted about a warning,

- If a handler will get control for the warning condition, the GET DIAGNOSTICS statement must be the first statement specified in that handler.
- If a handler will not get control for the warning condition, the GET DIAGNOSTICS statement must be the next statement executed after that previous statement.

SQL-variable-name-4 **or** *SQL-parameter-name-4*
Identifies a variable described in the program in accordance with the rules for declaring SQL variables or SQL parameters. The data type of the SQL variable or SQL parameter must be VARCHAR. The SQL variable or SQL parameter is assigned according to the retrieval assignment rules described in "Retrieval assignment" on page 102. If the length of *SQL-variable-name-4* or *SQL-parameter-name-4* is not sufficient to hold the full returned diagnostic string, the string is truncated, a warning is returned (SQLSTATE 01004) and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition.

ALL
Indicates that all diagnostic items that are set for the last SQL statement executed should be combined into one string. The format of the string is a semicolon separated list of all of the available diagnostic information in the form:

```
item-name=character-form-of-the-item-value;
```

The character form of a positive numeric value will not contain a leading plus sign (+) unless the item is RETURNED_SQLCODE. In this case, a leading plus sign (+) is added. For example:

```
NUMBER=1;RETURNED_SQLSTATE=02000;DB2_RETURNED_SQLCODE=+100;
```

Only items that contain diagnostic information are included in the string. There are also no entries in this string for the DB2_GET_DIAGNOSTICS_DIAGNOSTICS and DB2_SQL_NESTING_LEVEL items.

STATEMENT
Indicates that all *statement-information-item* diagnostic items that contain diagnostic information for the last SQL statement executed should be combined into one string. The format is the same as described above for ALL.

CONDITION
Indicates that *condition-information-item* diagnostic items that contain diagnostic information for the last SQL statement executed should be combined into one string. If *SQL-variable-name-5* or *SQL-parameter-name-5* or *integer* is specified, then the format is the same as described above for the ALL option. If *SQL-variable-name-5* or *SQL-parameter-name-5* or *integer* is not specified, then the format includes a condition number entry at the beginning of the information for that condition in the form:

```
CONDITION_NUMBER=X;item-name=character-form-of-the-item-value;
```

where X is the number of the condition. For example:

```
CONDITION_NUMBER=1;RETURNED_SQLSTATE=02000;RETURNED_SQLCODE=+100;
CONDITION_NUMBER=2;RETURNED_SQLSTATE=01004;
```

CONNECTION
Indicates that *connection-information-item* diagnostic items that contain diagnostic information for the last SQL statement executed should be combined into one string. If *SQL-variable-name-5* or *SQL-parameter-name-5* or *integer* is specified, then the format is the same as described above for ALL. If *SQL-variable-name-5* or *SQL-parameter-name-5* or *integer* is not specified, then the format includes a connection number entry at the beginning of the information for that condition in the form:

```
CONNECTION_NUMBER=X;item-name=character-form-of-the-item-value;
```

where X is the number of the condition. For example:

```
CONNECTION_NUMBER=1;CONNECTION_NAME=SVL1;DB2_PRODUCT_ID=DSN07010;
```

SQL-variable-name-5 **or** *SQL-parameter-name-5* **or** *integer*
Identifies the diagnostic for which ALL CONDITION or ALL CONNECTION information is requested. The SQL variable or SQL parameter specified must be described in the program in accordance with the rules for declaring integer SQL variables or SQL parameters. The value specified must not be less than one or greater than the number of available diagnostics.

statement-information-item
For a description of the *statement-information-items*, see "statement-information-item" on page 1199.

connection-information-item
For a description of the *connection-information-items*, see "connection-information-item" on page 1203.

condition-information-item
For a description of the *condition-information-items*, see "condition-information-item" on page 1204.

Notes

Effect of statement:: The GET DIAGNOSTICS statement does not change the contents of the diagnostics area except for DB2_GET_DIAGNOSTICS_DIAGNOSTICS.

Considerations for the SQLCODE and SQLSTATE SQL variables: The GET DIAGNOSTICS statement does not change the value of the SQLSTATE and SQLCODE SQL variables.

Case of return values: Values for identifiers in returned diagnostic items are not delimited and are case sensitive. For example, a table name of "abc" would be returned, simply as abc.

Data types for items: When a diagnostic item is assigned to a SQL variable or SQL parameter, the SQL variable or SQL parameter must be compatible with the data type of the diagnostic item. For more information, see Table 85 on page 1211.

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword EXCEPTION can be used as a synonym for CONDITION.
- The keyword RETURN_STATUS can be used as a synonym for DB2_RETURN_STATUS.

Example

Example 1:: In an SQL procedure, execute a GET DIAGNOSTICS statement to determine how many rows were updated.

```
CREATE PROCEDURE sqlprocg (IN deptnbr VARCHAR(3))
  LANGUAGE SQL
  BEGIN
    DECLARE SQLSTATE CHAR(5);
    DECLARE rcount INTEGER;
    UPDATE CORPDATA.PROJECT
      SET PRSTAFF = PRSTAFF + 1.5
      WHERE DEPTNO = deptnbr;
    GET DIAGNOSTICS rcount = ROW_COUNT;
    /* At this point, rcount contains the number of rows that were updated. */
  END
```

Example 2:: Within an SQL procedure, handle the returned status value from the invocation of a stored procedure called TRYIT. TRYIT could use the RETURN statement to explicitly return a status value or a status value could be implicitly returned by the database manager. If the procedure is successful, it returns a value of zero.

```
CREATE PROCEDURE TESTIT ()
  LANGUAGE SQL
  A1: BEGIN
      DECLARE RETVAL INTEGER DEFAULT 0;
      ...
      CALL TRYIT
      GET DIAGNOSTICS RETVAL = RETURN_STATUS;
      IF RETVAL <> 0 THEN
         ...
         LEAVE A1;
```

```
      ELSE
         ...
      END IF;
   END A1
```

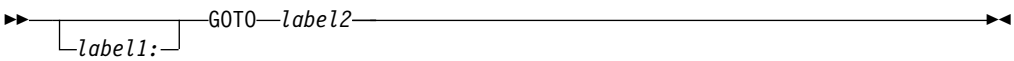
Example 3:: In an SQL procedure, execute a GET DIAGNOSTICS statement to retrieve the message text for an error.

```
CREATE PROCEDURE divide2 ( IN numerator INTEGER,
                           IN denominator INTEGER,
                          OUT divide_result INTEGER,
                              OUT divide_error VARCHAR(70) )
LANGUAGE SQL
  BEGIN
     DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        GET DIAGNOSTICS EXCEPTION 1
          divide_error = MESSAGE_TEXT;
     SET divide_result = numerator / denominator;
  END;
```

GOTO statement

The GOTO statement branches to a user-defined label within an SQL function, SQL procedure, or SQL trigger.

Syntax



Description

label1
Specifies the label for the GOTO statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

label2
Specifies the labelled statement where processing is to continue. The labelled statement and the GOTO statement must both be in the same scope:

- If the GOTO statement is defined in a FOR statement, *label2* must be defined inside the same FOR statement, excluding a nested FOR statement or nested compound statement.
- If the GOTO statement is defined outside a FOR statement, *label2* must not be defined within a FOR statement or nested compound statement.
- If the GOTO statement is defined in a condition handler, *label2* must be defined inside the same handler.
- If the GOTO statement is defined outside a condition handler, *label2* must not be defined within a condition handler.

If *label2* is not defined within a scope that the GOTO statement can reach, an error is returned.

Notes

Using a GOTO statement: It is recommended that the GOTO statement be used sparingly. This statement interferes with normal sequence of processing SQL statements, thus making a routine more difficult to read and maintain. Before using a GOTO statement, determine whether another statement, such as IF or LEAVE, can be used in place, to eliminate the need for a GOTO statement.

Effect on open cursors: When a GOTO statement transfers control out of a compound statement, all open cursors that are declared in the compound statement that contains the GOTO statement are closed, unless they are declared to return result sets or unless *ENDACTGRP is specified.

Effect on ATOMIC compound statements: When a GOTO statement transfers control out of an ATOMIC compound statement, the savepoint that was implicitly started when the ATOMIC compound statement was entered is released.

Considerations for SQLSTATE and SQLCODE variables: The GOTO statement does not affect the SQLSTATE and SQLCODE SQL variables. At the end of the GOTO statement the SQLSTATE and SQLCODE SQL variables reflect the result of the last statement executed before that GOTO statement.

Example

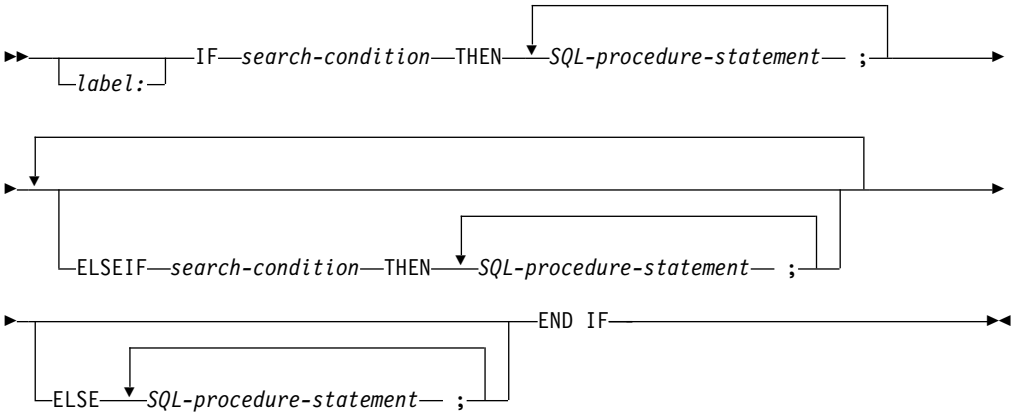
In the following statement, the parameters *rating* and *v_empno* are passed in to the procedure. The time in service is returned as a date duration in output parameter *return_parm*. If the time in service with the company is less then 6 months, the GOTO statement transfers control to the end of the procedure and *new_salary* is left unchanged.

```
CREATE PROCEDURE adjust_salary
  (IN v_empno CHAR(6),
   IN rating INTEGER,
   OUT return_parm DECIMAL(8,2))
  LANGUAGE SQL
  MODIFIES SQL DATA
    BEGIN
       DECLARE new_salary DECIMAL(9,2);
       DECLARE service DECIMAL(8,2);
            SELECT salary, CURRENT_DATE - hiredate
               INTO new_salary, service
               FROM employee
               WHERE empno = v_empno;
            IF service < 600
              THEN GOTO exit1;
            END IF;
            IF rating = 1
             THEN SET new_salary = new_salary + (new_salary * .10);
             ELSEIF rating = 2
             THEN SET new_salary = new_salary + (new_salary * .05);
            END IF;
            UPDATE employee
               SET salary = new_salary
               WHERE empno = v_empno;

       EXIT1: SET return_parm = service;
     END
```

IF statement

The IF statement executes different sets of SQL statements based on the result of search conditions.

Syntax



Description

label
Specifies the label for the IF statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

search-condition
Specifies the *search-condition* for which an SQL statement should be executed. If the condition is unknown or false, processing continues to the next search condition, until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement
Specifies an SQL statement to execute if the preceding *search-condition* is true.

Notes

Considerations for SQLSTATE and SQLCODE SQL variables: When the first *SQL-procedure-statement* in the IF statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the *search-conditions* of that IF statement. If an IF statement does not include an ELSE clause and none of the *search-conditions* evaluate to true, then when the statement that follows the IF statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the search conditions of that IF statement.

Example

The following SQL procedure accepts two IN parameters: an employee number and an employee rating. Depending on the value of *rating*, the employee table is updated with new values in the salary and bonus columns.

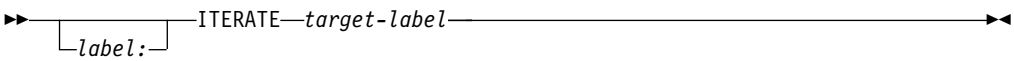
```
CREATE PROCEDURE UPDATE_SALARY_IF
   (IN employee_number CHAR(6), INOUT rating SMALLINT)
   LANGUAGE SQL
```

```
MODIFIES SQL DATA
BEGIN
   DECLARE not_found CONDITION FOR SQLSTATE '02000';
   DECLARE EXIT HANDLER FOR not_found
      SET rating = -1;
   IF rating = 1
      THEN UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
   ELSEIF rating = 2
      THEN UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
   ELSE UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
   END IF;
END
```

ITERATE statement

The ITERATE statement causes the flow of control to return to the beginning of a labelled loop.

Syntax



Description

label
Specifies the label for the ITERATE statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

target-label
Specifies the label of the FOR, LOOP, REPEAT, or WHILE statement to which the flow of control is passed. *target-label* must be defined as a label for a FOR, LOOP, REPEAT, or WHILE statement. The ITERATE statement must be in that FOR, LOOP, REPEAT, or WHILE statement, or in the block of code that is directly or indirectly nested within that statement, subject to the following restrictions:

- If the ITERATE statement is in a condition handler, *target-label* must be defined in that condition handler.
- If the ITERATE statement is not in a condition handler, *target-label* must not be defined in a condition handler.
- If the ITERATE statement is in a FOR statement, *target-label* must be that label on that FOR statement, or the label must be defined inside that FOR statement.

Notes

Considerations for SQLSTATE and SQLCODE variables: The ITERATE statement does not affect the SQLSTATE and SQLCODE SQL variables. At the end of the ITERATE statement the SQLSTATE and SQLCODE SQL variables reflect the result of the last statement executed before that ITERATE statement.

Example

This example uses a cursor to return information for a new department. If the *not_found* condition handler was invoked, the flow of control passes out of the loop. If the value of *v_dept* is 'D11', an ITERATE statement passes the flow of control back to the top of the LOOP statement. Otherwise, a new row is inserted into the DEPARTMENT table.

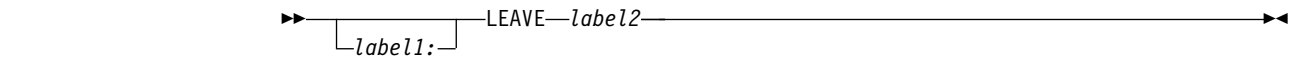
```
CREATE PROCEDURE ITERATOR ()
   LANGUAGE SQL
   MODIFIES SQL DATA
   BEGIN
      DECLARE v_dept CHAR(3);
      DECLARE v_deptname VARCHAR(29);
      DECLARE v_admdept CHAR(3);
      DECLARE at_end INTEGER DEFAULT 0;
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE c1 CURSOR FOR
        SELECT deptno,deptname,admrdept
```

```
      FROM department
      ORDER BY deptno;
   DECLARE CONTINUE HANDLER FOR not_found
      SET at_end = 1;
   OPEN c1;
   ins_loop:
   LOOP
      FETCH c1 INTO v_dept, v_deptname, v_admdept;
      IF at_end = 1 THEN
         LEAVE ins_loop;
      ELSEIF v_dept ='D11' THEN
         ITERATE ins_loop;
      END IF;
      INSERT INTO department (deptno,deptname,admrdept)
         VALUES('NEW', v_deptname, v_admdept);
   END LOOP;
   CLOSE c1;
END
```

LEAVE statement

The LEAVE statement continues execution by leaving a block or loop.

Syntax



Description

label1
Specifies the label for the LEAVE statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

label2
Specifies the label of the compound, FOR, LOOP, REPEAT, or WHILE statement to exit.

The LEAVE statement cannot be used to leave a handler.

Notes

Effect on open cursors: When a LEAVE statement transfers control out of a compound statement, all open cursors that are declared in the compound statement that contains the LEAVE statement are closed, unless they are declared to return result sets or unless *ENDACTGRP is specified.

Considerations for SQLSTATE and SQLCODE variables: The LEAVE statement does not affect the SQLSTATE and SQLCODE SQL variables. At the end of the LEAVE statement the SQLSTATE and SQLCODE SQL variables reflect the result of the last statement executed before that LEAVE statement.

Examples

The example contains a loop that fetches data for cursor *c1*. If the value of SQL variable *at_end* is not zero, the LEAVE statement transfers control out of the loop.

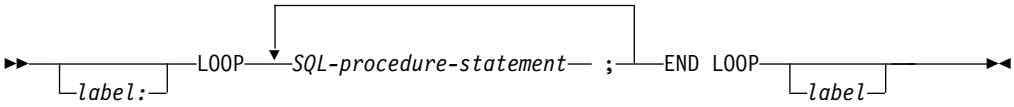
```
CREATE PROCEDURE LEAVE_LOOP (OUT COUNTER INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER;
      DECLARE v_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE at_end SMALLINT DEFAULT 0;
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
         FROM employee;
      DECLARE CONTINUE HANDLER FOR not_found
         SET at_end = 1;
      SET v_counter = 0;
      OPEN c1;
      fetch_loop:
      LOOP
         FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
         IF at_end <> 0 THEN
            LEAVE fetch_loop;
         END IF;
```

```
      SET v_counter = v_counter + 1;
   END LOOP fetch_loop;
   SET counter = v_counter;
   CLOSE c1;
END
```

LOOP statement

The LOOP statement repeats the execution of a statement or a group of statements.

Syntax



Description

label
Specifies the label for the LOOP statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

SQL-procedure statement
Specifies an SQL statement to be executed in the loop

Notes

Considerations for the diagnostics area: At the beginning of the first iteration of the LOOP statement, and with every subsequent iteration, the diagnostics area is cleared.

Considerations for SQLSTATE and SQLCODE variables: Prior to executing the first *SQL-procedure statement* within that LOOP statement, the SQLSTATE and SQLCODE values reflect the last values that were set prior to the LOOP statement. If the loop is terminated with a GOTO or a LEAVE statement, the SQLSTATE and SQLCODE values reflect the successful completion of that statement. When the LOOP statement iterates, the SQLSTATE and SQLCODE values reflect the result of the last SQL statement executed within the LOOP statement.

Examples

This procedure uses a LOOP statement to fetch values from the employee table. Each time the loop iterates, the OUT parameter *counter* is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes the flow of control outside of the loop.

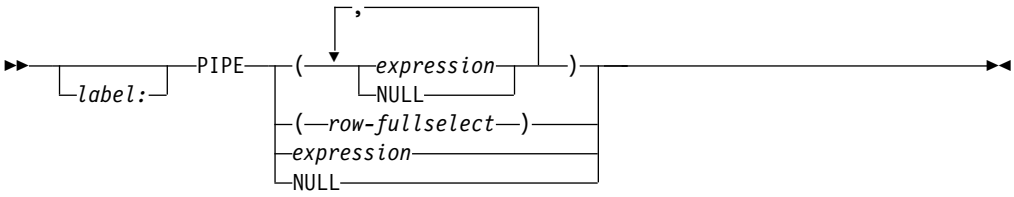
```
CREATE PROCEDURE LOOP_UNTIL_SPACE (OUT COUNTER INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER DEFAULT 0;
      DECLARE v_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
         FROM employee;
      DECLARE CONTINUE HANDLER FOR NOT FOUND
         SET counter = -1;
      OPEN c1;
      fetch_loop:
      LOOP
```

```
      FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
      IF v_midinit = ' ' THEN
         LEAVE fetch_loop;
      END IF;
      SET v_counter = v_counter + 1;
   END LOOP fetch_loop;
   SET counter = v_counter;
   CLOSE c1;
END
```

PIPE statement

The PIPE statement returns one row from a table function. An SQL table function that uses a PIPE statement is referred to as a pipelined function.

Syntax



Description

label
Specifies the label for the PIPE statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

(*expression***, ...)**
Specifies a row value is returned from the function. The number of expressions or NULL keywords in the list must match the number of columns in the function result. The data types of each expression must be assignable to the data type of the corresponding column defined for the function result, using the storage assignment rules as described in "Assignments and comparisons" on page 98.

row-fullselect
Specifies a fullselect that returns a single row. The number of columns in the fullselect must match the number of columns in the function result. The data types of the result table columns of the fullselect must be assignable to the data types of the columns defined for the function result, using the storage assignment rules as described in "Assignments and comparisons" on page 98. If the result of the row fullselect is no rows, a null value is returned for each column. An error is returned if there is more than one row in the result.

expression
Specifies a scalar value is returned from the function. The table function must return a single column and the value of the expression must be assignable to that column.

NULL
Specifies that a null value is returned from the function. If there is more than one result column, a null value is returned for each column.

Example

Use a PIPE statement to return rows from an SQL table function.

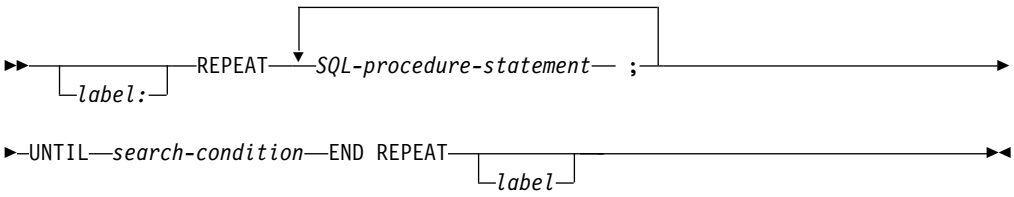
```
CREATE FUNCTION TRANSFORM() RETURNS TABLE ( EMPLOYEE_NAME CHAR(20), UNIQUE# INT )
BEGIN
  DECLARE EMPNAME VARCHAR(15);
  DECLARE MYRECNUM INTEGER DEFAULT 1;
  DECLARE AT_END INTEGER DEFAULT 0;
  DECLARE EMP_CURSOR CURSOR FOR SELECT lastname FROM employee;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
    SET AT_END = 1;
```

```
|
|                         OPEN EMP_CURSOR;
|                         MYLOOP: LOOP
|                           FETCH EMP_CURSOR INTO EMPNAME;
|                           IF AT_END = 1 THEN
|                             LEAVE MYLOOP;
|                           END IF;
|                           PIPE (EMPNAME, MYRECNUM);  -- return single row
|                           SET MYRECNUM = MYRECNUM + 1;
|                         END LOOP;
|
|                         CLOSE EMP_CURSOR;
|                         RETURN;
|                       END;
```

REPEAT statement

The REPEAT statement executes a statement or group of statements until a search condition is true.

Syntax



Description

label
Specifies the label for the REPEAT statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

SQL-procedure-statement
Specifies an SQL statement to be executed in the REPEAT loop.

search-condition
The *search-condition* is evaluated after each execution of the REPEAT loop. If the condition is true, the REPEAT loop will exit. If the condition is unknown or false, the looping continues.

Notes

Considerations for the diagnostics area: At the beginning of the first iteration of the REPEAT statement, and with every subsequent iteration, the diagnostics area is cleared.

Considerations for SQLSTATE and SQLCODE SQL variables: With each iteration of the REPEAT statement, the SQLSTATE and SQLCODE SQL variables are cleared prior to executing the first *SQL-procedure-statement* within the REPEAT statement. At the beginning of the first iteration of the REPEAT statement, the SQLSTATE and SQLCODE values reflect the last values that were set prior to the REPEAT statement. At the beginning of iterations 2 through n of the REPEAT statement, the SQLSTATE and SQLCODE values reflect the result of evaluating the search condition in the UNTIL clause of that REPEAT statement. If the loop is terminated with a GOTO, ITERATE, or LEAVE statement, the SQLSTATE and SQLCODE values reflect the successful completion of that statement. Otherwise, after the END REPEAT of the REPEAT statement completes, the SQLSTATE and SQLCODE values reflect the result of evaluating the search condition in the UNTIL clause of that REPEAT statement.

Example

A REPEAT statement fetches rows from a table until the *not_found* condition handler is invoked.

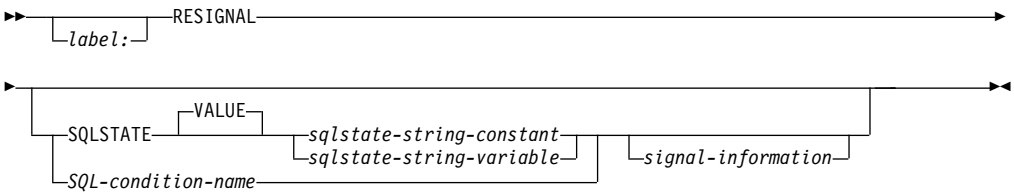
```
CREATE PROCEDURE REPEAT_STMT (OUT COUNTER INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER DEFAULT 0;
      DECLARE v_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE at_end SMALLINT DEFAULT 0;
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
         FROM employee;
      DECLARE CONTINUE HANDLER FOR not_found
         SET at_end = 1;
      OPEN c1;
      fetch_loop:
      REPEAT
         FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
         SET v_counter = v_counter + 1;
         UNTIL at_end > 0
      END REPEAT fetch_loop;
      SET counter = v_counter;
      CLOSE c1;
   END
```

RESIGNAL statement

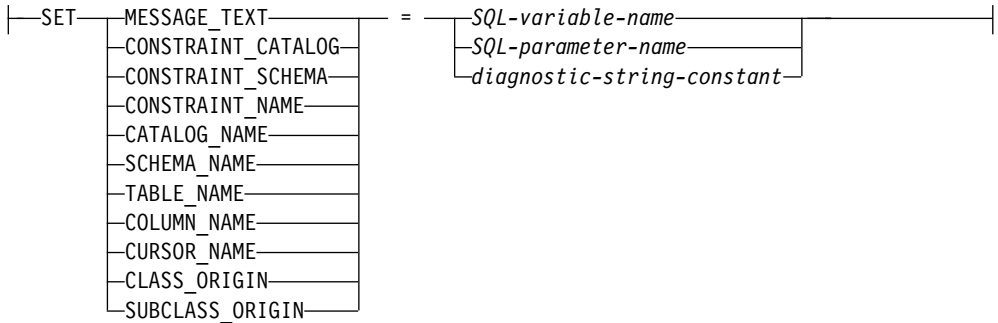
The RESIGNAL statement is used within a condition handler to re-raise the current condition, or raise an alternate condition so that it can be processed at a higher level. It causes an exception, warning, or not found condition to be returned, along with optional message text.

Issuing the RESIGNAL statement without an operand causes the current condition to be passed outwards.

Syntax



signal-information:



Description

label
Specifies the label for the RESIGNAL statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

SQLSTATE VALUE
Specifies the SQLSTATE that will be resignaled. Any valid SQLSTATE value can be used. The specified value must follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00' since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is returned.

sqlstate-string-constant
The *sqlstate-string-constant* must be a character string constant with exactly 5 characters.

sqlstate-string-variable
The *sqlstate-string-variable* must be a character or Unicode graphic variable. The actual length of the contents of the *sqlstate-string-variable* must be 5.

SQL-condition-name
Specifies the name of the condition that will be returned. The *SQL-condition-name* must be declared within the *compound-statement*.

SET
Introduces the assignment of values to *condition-information-items*. The *condition-information-item* values can be accessed using the GET DIAGNOSTICS statement. The only *condition-information-item* that can be accessed in the SQLCA is MESSAGE_TEXT.

MESSAGE_TEXT
Specifies a string that describes the error or warning.

If an SQLCA is used,

- the string is returned in the SQLERRMC field of the SQLCA
- if the actual length of the string is longer than 1000 bytes, it is truncated without a warning.

CONSTRAINT_CATALOG
Specifies a string that indicates the name of the database that contains a constraint related to the signalled error or warning.

CONSTRAINT_SCHEMA
Specifies a string that indicates the name of the schema that contains a constraint related to the signalled error or warning.

CONSTRAINT_NAME
Specifies a string that indicates the name of a constraint related to the signalled error or warning.

CATALOG_NAME
Specifies a string that indicates the name of the database that contains a table or view related to the signalled error or warning.

SCHEMA_NAME
Specifies a string that indicates the name of the schema that contains a table or view related to the signalled error or warning.

TABLE_NAME
Specifies a string that indicates the name of a table or view related to the signalled error or warning.

COLUMN_NAME
Specifies a string that indicates the name of a column in the table or view related to the signalled error or warning.

CURSOR_NAME
Specifies a string that indicates the name of a cursor related to the signalled error or warning.

CLASS_ORIGIN
Specifies a string that indicates the origin of the SQLSTATE class related to the signalled error or warning.

SUBCLASS_ORIGIN
Specifies a string that indicates the origin of the SQLSTATE subclass related to the signalled error or warning.

SQL-variable-name
Identifies an SQL variable declared within the *compound-statement*, that

contains the value to be assigned to the *condition-information-item*. The SQL variable must be defined as CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable.

SQL-parameter-name
Identifies an SQL parameter declared within the *compound-statement*, that contains the value to be assigned to the *condition-information-item*. The SQL parameter must be defined as CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable.

diagnostic-string-constant
Specifies a character string constant that contains the value to be assigned to the *condition-information-item*.

Notes

SQLSTATE values: Any valid SQLSTATE value can be used in the RESIGNAL statement. However, it is recommended that programmers define new SQLSTATEs based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

For more information about SQLSTATEs, see the SQL Messages and Codes topic collection.

Assignment: When the RESIGNAL statement is executed, the value of each of the specified *string-constants* and *variables* is assigned to the corresponding *condition-information-item*. However, if the length of a *string-constant* or *variable* is longer than the maximum length of the corresponding *condition-information-item*, it is truncated without a warning. For details on the assignment rules, see "Assignments and comparisons" on page 98. For details on the maximum length of specific *condition-information-items*, see "GET DIAGNOSTICS" on page 1194.

Processing a RESIGNAL statement:

- If the RESIGNAL statement is specified without a SQLSTATE clause or a *SQL-condition-name*, the SQL function, SQL procedure, or SQL trigger resignals the identical condition that invoked the handler and the SQLCODE is not changed.
- When a RESIGNAL statement is issued and an SQLSTATE or *SQL-condition-name* is specified, the SQLCODE is based on the SQLSTATE value as follows:
  - If the specified SQLSTATE class is either '01' or '02', a warning or not found is signalled and the SQLCODE is set to +438.
  - Otherwise, an exception is returned and the SQLCODE is set to –438.

If the SQLSTATE or condition indicates that an exception is signalled (SQLSTATE class other than '01' or '02'):,

- If a handler exists in the same compound statement as the RESIGNAL statement, and the *compound-statement* contains a handler for SQLEXCEPTION or the specified SQLSTATE or condition; the exception is handled and control is transferred to that handler.
- If the *compound-statement* is nested and an outer level *compound-statement* has a handler for SQLEXCEPTION or the specified SQLSTATE or condition; the exception is handled and control is transferred to that handler.
- Otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.

If the SQLSTATE or condition indicates that a warning (SQLSTATE class '01') or not found (SQLSTATE class '02') is signalled:

- If a handler exists in the same compound statement as the RESIGNAL statement, and the *compound-statement* contains a handler for SQLWARNING (if the SQLSTATE class is '01'), NOT FOUND (if the SQLSTATE class is '02'), or the specified SQLSTATE or condition; the warning or not found condition is handled and control is transferred to that handler.
- If the *compound-statement* is nested and an outer level compound statement contains a handler for SQLWARNING (if the SQLSTATE class is '01'), NOT FOUND (if the SQLSTATE class is '02'), or the specified SQLSTATE or condition; the warning or not found condition is not handled and processing continues with the next statement.
- Otherwise, the warning is not handled and processing continues with the next statement.

Considerations for the diagnostics area: The RESIGNAL statement might modify the contents of the current diagnostics area. If an SQLSTATE or *SQL-condition-name* is specified as part of the RESIGNAL statement, the RESIGNAL statement starts with a clear of the diagnostics area and sets the RETURNED_SQLSTATE to reflect the specified SQLSTATE or *SQL-condition-name*. If any *signal-information* is specified, the corresponding items in the condition area are assigned the specified values. DB2_RETURNED_SQLCODE is set to +438 or -438 corresponding to the specified SQLSTATE or *SQL-condition-name*.

Example

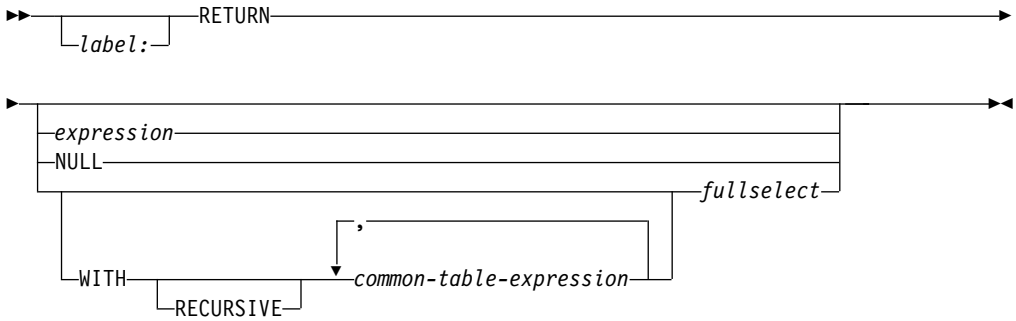
This example detects a division by zero error. The IF statement uses a SIGNAL statement to invoke the overflow condition handler. The condition handler uses a RESIGNAL statement to return a different SQLSTATE value to the client application.

```
CREATE PROCEDURE divide ( IN numerator INTEGER,
                          IN denominator INTEGER,
                           OUT divide_result INTEGER )
  LANGUAGE SQL
    BEGIN
       DECLARE overflow CONDITION FOR '22003';
       DECLARE CONTINUE HANDLER FOR overflow
           RESIGNAL SQLSTATE '22375';
       IF denominator = 0 THEN
           SIGNAL overflow;
       ELSE
           SET divide_result = numerator / denominator;
       END IF;
     END
```

RETURN statement

The RETURN statement returns from a routine. For SQL functions, it returns the result of the function. For SQL non-pipelined table functions, it returns a table as the result of the function. For SQL pipelined table functions, it indicates the end of the result table has been reached. For an SQL procedure, it optionally returns an integer status value.

Syntax



Description

label
Specifies the label for the RETURN statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

expression
Specifies a value that is returned from the routine:

- If the routine is a scalar function, the data type of the result must be assignable to the data type defined for the function result, using the storage assignment rules as described in "Assignments and comparisons" on page 98. An aggregate function, or user-defined function that is sourced on an aggregate function must not be specified for a RETURN statement in an SQL scalar function.
- If the routine is a table function, a scalar expression (other than a scalar fullselect) cannot be specified.
- If the routine is a procedure, the data type of *expression* must be INTEGER. If the *expression* evaluates to the null value, a value of zero is returned.

NULL
Specifies that the null value is returned from the routine.

- If the routine is a scalar function, the null value is returned.
- If the routine is a table function, NULL must not be specified.
- If the routine is a procedure, NULL must not be specified.

WITH *common-table-expression*
Specifies one or more *common table expressions* to be used in the fullselect.

fullselect
Specifies the row or rows to be returned for the routine.

- If the routine is a scalar function, the fullselect must return one column and, at most, one row. The data type of the result column must be assignable to

the data type defined for the function result, using the storage assignment rules as described in "Assignments and comparisons" on page 98.

- If the routine is a table function, the fullselect can return zero or more rows with one or more columns. The number of columns in the fullselect must match the number of columns in the function result. In addition, the data types of the result table columns of the fullselect must be assignable to the data types of the columns defined for the function result, using the storage assignment rules as described in "Assignments and comparisons" on page 98.
- If the routine is a procedure, fullselect must not be specified.

Notes

Returning from a table function:

- The last statement executed in the *SQL-routine-body* must be a RETURN statement.
- For a non-pipelined table function, the RETURN statement indicates the rows to be returned. Exactly one RETURN statement must be specified and it must contain a *fullselect*.
- For a pipelined table function, one or more RETURN statements must be specified and they must contain no return values.

Returning from a procedure:

- If a RETURN statement with a specified return value is used to return from a procedure then the SQLCODE, SQLSTATE, and message length in the SQLCA or diagnostics area are initialized to zeros, and message text is set to blanks. An error is not returned to the caller.
- If a RETURN statement is not used to return from a procedure or if a value is not specified on the RETURN statement,
  - if the procedure returns with an SQLCODE that is greater than or equal to zero, the specified target for DB2_RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of 0
  - if the procedure returns with an SQLCODE that is less than zero, the specified target for DB2_RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of -1.
- When a value is returned from a procedure, the caller may access the value using:
  - the GET DIAGNOSTICS statement to retrieve the DB2_RETURN_STATUS when the SQL procedure was called from another SQL procedure
  - the parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a ODBC or JDBC application
  - directly from the SQLCA returned from processing the CALL of an SQL procedure by retrieving the value of sqlerrd[0] when the SQLCODE is not less than zero. When the SQLCODE is less than zero, the sqlerrd[0] value is not set and the application should assume a return status value of -1.

RETURN restrictions:

- RETURN is not allowed in SQL triggers.
- In a compound(dynamic) statement, RETURN cannot specify a return value.

Example

Example 1: Use a RETURN statement to return from an SQL procedure with a status value of zero if successful, and –200 if not.

```
BEGIN
  ...
  GOTO fail;
  ...
  success: RETURN 0
  failure: RETURN -200
...
END
```

Example 2: Define a scalar function that returns the tangent of a value using the existing sine and cosine functions.

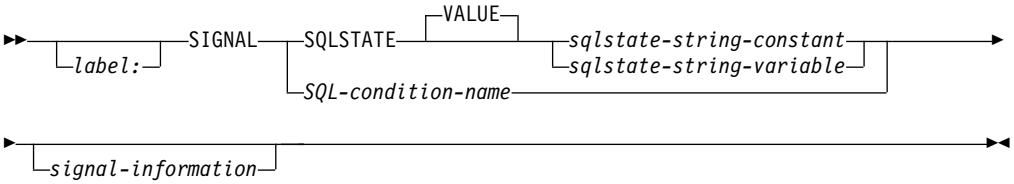
```
CREATE FUNCTION mytan (x DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(x)/COS(x)
```

SIGNAL statement

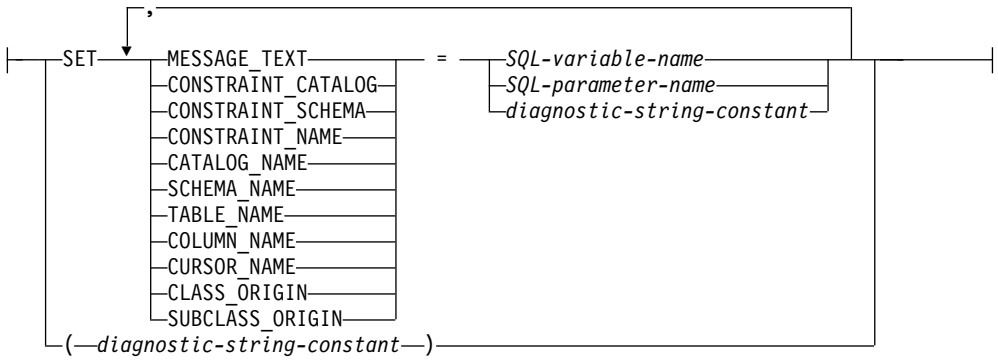
The SIGNAL statement signals an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE and optional *condition-information-items*. The syntax of SIGNAL in an SQL function, SQL procedure, or SQL trigger is a similar to what is supported as a SIGNAL statement in other contexts.

See "SIGNAL" on page 1407 for details.

Syntax



signal-information:



Description

label
Specifies the label for the SIGNAL statement. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

SQLSTATE VALUE
Specifies the SQLSTATE that will be signalled. The specified value must not be null and must follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00' since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is returned.

sqlstate-string-constant
The *sqlstate-string-constant* must be a character string constant with exactly 5 characters.

sqlstate-string-variable
The *sqlstate-string-variable* must be a character or Unicode graphic variable. The actual length of the contents of the *variable* must be 5.

SQL-condition-name
Specifies the name of the condition that will be signalled. The *SQL-condition-name* must be declared within the *compound-statement*.

SET
Introduces the assignment of values to *condition-information-items*. The *condition-information-item* values can be accessed using the GET DIAGNOSTICS statement. The only *condition-information-item* that can be accessed in the SQLCA is MESSAGE_TEXT.

MESSAGE_TEXT
Specifies a string that describes the error or warning.

If an SQLCA is used,

- the string is returned in the SQLERRMC field of the SQLCA
- if the actual length of the string is longer than 1000 bytes, it is truncated without a warning.

CONSTRAINT_CATALOG
Specifies a string that indicates the name of the database that contains a constraint related to the signalled error or warning.

CONSTRAINT_SCHEMA
Specifies a string that indicates the name of the schema that contains a constraint related to the signalled error or warning.

CONSTRAINT_NAME
Specifies a string that indicates the name of a constraint related to the signalled error or warning.

CATALOG_NAME
Specifies a string that indicates the name of the database that contains a table or view related to the signalled error or warning.

SCHEMA_NAME
Specifies a string that indicates the name of the schema that contains a table or view related to the signalled error or warning.

TABLE_NAME
Specifies a string that indicates the name of a table or view related to the signalled error or warning.

COLUMN_NAME
Specifies a string that indicates the name of a column in the table or view related to the signalled error or warning.

CURSOR_NAME
Specifies a string that indicates the name of a cursor related to the signalled error or warning.

CLASS_ORIGIN
Specifies a string that indicates the origin of the SQLSTATE class related to the signalled error or warning.

SUBCLASS_ORIGIN
Specifies a string that indicates the origin of the SQLSTATE subclass related to the signalled error or warning.

SQL-variable-name
Identifies an SQL variable declared within the *compound-statement*, that contains the value to be assigned to the *condition-information-item*. The SQL variable must be defined as CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable.

SQL-parameter-name
Identifies an SQL parameter declared within the *compound-statement*, that contains the value to be assigned to the *condition-information-item*. The SQL parameter must be defined as CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC variable.

diagnostic-string-constant
Specifies a character string constant that contains the value to be assigned to the *condition-information-item*.

(diagnostic-string-constant)
Specifies a character string constant that contains the message text.

This form is only allowed in the triggered action of a CREATE TRIGGER statement.

To conform with the ANS and ISO standards, this form should not be used. It is provided for compatibility with other products.

Notes

SQLSTATE values: Any valid SQLSTATE value can be used in the SIGNAL statement. However, it is recommended that programmers define new SQLSTATEs based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

For more information about SQLSTATEs, see the SQL Messages and Codes topic collection.

Assignment: When the SIGNAL statement is executed, the value of each of the specified *string-constants* and *variables* is assigned to the corresponding *condition-information-item*. However, if the length of a *string-constant* or *variable* is longer than the maximum length of the corresponding *condition-information-item*, it is truncated without a warning. For details on the assignment rules, see "Assignments and comparisons" on page 98. For details on the maximum length of specific *condition-information-items*, see "GET DIAGNOSTICS" on page 1194.

Processing a SIGNAL statement: When a SIGNAL statement is issued, the SQLCODE returned in the SQLCA is based on the SQLSTATE value as follows:

- If the specified SQLSTATE class is either '01' or '02', a warning or not found is signalled and the SQLCODE is set to +438.
- Otherwise, an exception is signalled and the SQLCODE is set to –438.

If the SQLSTATE or condition indicates that an exception (SQLSTATE class other than '01' or '02') is signalled,

- If a handler exists in the same compound statement as the SIGNAL statement, and the compound statement contains a handler for SQLEXCEPTION or the specified SQLSTATE or condition; the exception is handled and control is transferred to that handler.

- If the *compound-statement* is nested and an outer level *compound-statement* has a handler for SQLEXCEPTION or the specified SQLSTATE or condition; the exception is handled and control is transferred to that handler.
- Otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.

If the SQLSTATE or condition indicates that a warning (SQLSTATE class '01') or not found (SQLSTATE class '02') is signalled,

- If a handler exists in the same compound statement as the SIGNAL statement, and the compound statement contains a handler for SQLWARNING (if the SQLSTATE class is '01'), NOT FOUND (if the SQLSTATE class is '02'), or the specified SQLSTATE or condition; the warning or not found condition is handled and control is transferred to that handler.
- If the *compound-statement* is nested and an outer level compound statement contains a handler for SQLWARNING (if the SQLSTATE class is '01'), NOT FOUND (if the SQLSTATE class is '02'), or the specified SQLSTATE or condition; the warning or not found condition is not handled and processing continues with the next statement.
- Otherwise, the warning is not handled and processing continues with the next statement.

Considerations for the diagnostics area: The SIGNAL statement starts with a clear of the diagnostics area and then sets the RETURNED_SQLSTATE to reflect the specified SQLSTATE or *SQL-condition-name*. If any *signal-information* is specified, the corresponding items in the condition area are assigned the specified values. DB2_RETURNED_SQLCODE is set to +438 or -438 corresponding to the specified SQLSTATE or *SQL-condition-name*.

Example

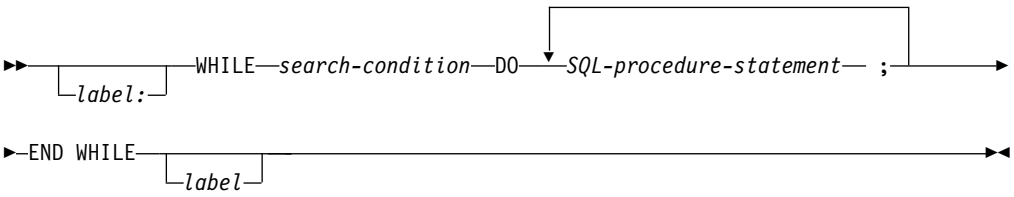
An SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
   (IN ONUM INTEGER, IN CNUM INTEGER,
    IN PNUM INTEGER, IN QNUM INTEGER)
    LANGUAGE SQL
    MODIFIES SQL DATA
   BEGIN
      DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
         SIGNAL SQLSTATE '75002'
            SET MESSAGE_TEXT = 'Customer number is not known';
      INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
         VALUES (ONUM, CNUM, PNUM, QNUM);
   END
```

WHILE statement

The WHILE statement repeats the execution of a statement while a specified condition is true.

Syntax



Description

label
Specifies the label for the WHILE statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to SQL labels" on page 1433.

search-condition
Specifies a condition that is evaluated before each execution of the WHILE loop. If the condition is true, the *SQL-procedure-statements* in the WHILE loop are executed.

SQL-procedure-statement
Specifies an SQL statement or statements to execute within the WHILE loop.

Notes

Considerations for the diagnostics area: At the beginning of the first iteration of the WHILE statement, and with every subsequent iteration, the diagnostics area is cleared.

Considerations for SQLSTATE and SQLCODE SQL variables: With each iteration of the WHILE statement, when the first *SQL-procedure-statement* is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the search condition of that WHILE statement. If the loop is terminated with a GOTO, ITERATE, or LEAVE statement, the SQLSTATE and SQLCODE values reflect the successful completion of that statement. Otherwise, after the END WHILE of the WHILE statement completes, the SQLSTATE and SQLCODE reflect the result of evaluating the search condition of that WHILE statement.

Example

This example uses a WHILE statement to iterate through FETCH and SET statements. While the value of SQL variable *v_counter* is less than half of number of employees in the department identified by the IN parameter *deptNumber*, the WHILE statement continues to perform the FETCH and SET statements. When the condition is no longer true, the flow of control leaves the WHILE statement and closes the cursor.

```
CREATE PROCEDURE dept_median
   (IN deptNumber SMALLINT,
    OUT medianSalary DECIMAL(7,2))
    LANGUAGE SQL
    BEGIN
       DECLARE v_numRecords INTEGER DEFAULT 1;
       DECLARE v_counter INTEGER DEFAULT 0;
       DECLARE c1 CURSOR FOR
          SELECT salary
          FROM staff
          WHERE dept = deptNumber
          ORDER BY salary;
       DECLARE EXIT HANDLER FOR NOT FOUND
          SET medianSalary = 6666;
       SET medianSalary = 0;
       SELECT COUNT(*) INTO v_numRecords
          FROM staff
          WHERE dept = deptNumber;
       OPEN c1;
       WHILE v_counter < (v_numRecords/2 + 1) DO
          FETCH c1 INTO medianSalary;
          SET v_counter = v_counter +1;
       END WHILE;
       CLOSE c1;
    END
```

Appendix A. SQL limits

The following tables describe certain SQL and database limits imposed by the DB2 for i database manager.

Note:

- System storage limits may preclude the limits specified here. For example, see "Maximum row sizes" on page 1027.
- A limit of *storage* means that the limit is dependent on the amount of storage available.
- A limit of *statement* means that the limit is dependent on the limit for the maximum length of a statement.

Table 111. Identifier Length Limits

Identifier Limits	DB2 for i Limit
Longest authorization name	10[122]
Longest correlation name	128
Longest cursor name	128
Longest descriptor name	128
Longest external program name (string form)	279[123]
Longest external program name (unqualified form)	10
Longest host identifier[124]	128
Longest package version-id	64
Longest partition name	10
Longest savepoint name	128
Longest schema name	128
Longest server name	18
Longest statement name	128
Longest SQL condition name	128
Longest SQL label	128
Longest unqualified alias name	128
Longest unqualified column name	128
Longest unqualified constraint name	128
Longest unqualified distinct type name	128
Longest unqualified function name	128
Longest unqualified global variable name	128
Longest unqualified index name	128
Longest unqualified nodegroup name	10
Longest unqualified package name	10
Longest unqualified procedure name	128
Longest unqualified sequence name	128
Longest unqualified specific name	128
Longest unqualified SQL parameter name	128
Longest unqualified SQL variable name	128
Longest unqualified system column name	10
Longest unqualified system object name	10
Longest unqualified system schema name	10
Longest unqualified table and view name	128
Longest unqualified trigger name	128
Longest unqualified XSR object name	128
Longest XML element name, attribute name, prefix name, or processing instruction name specified in XMLELEMENT, XMLFOREST, XMLATTRIBUTES, XMLNAMESPACES or XMLPI	128
Longest XML element name, attribute name, prefix name, or processing instruction name for a parsed XML document	1000
Longest XML schema location uniform resource identifier (URI)	1000

122. As an application requester, DB2 for i can send an authorization name of up to 255 bytes.

123. For REXX procedures, the limit is 33.

124. For an RPG, COBOL, or REXX program, the limit is 64.

Table 112. Numeric Limits

Numeric Limits	DB2 for i Limit
Smallest SMALLINT value	-32 768
Largest SMALLINT value	+32 767
Smallest INTEGER value	-2 147 483 648
Largest INTEGER value	+2 147 483 647
Smallest BIGINT value	-9 223 372 036 854 775 808
Largest BIGINT value	+9 223 372 036 854 775 807
Largest decimal precision	63
Maximum exponent (E^{max}) for REAL values	38
Smallest REAL value[125]	$-3.4x10^{38}$
Largest REAL value[125]	$+3.4x10^{38}$
Minimum exponent (E^{min}) for REAL values	-38
Smallest positive REAL value[125]	$+1.18x10^{-38}$
Largest negative REAL value[125]	$-1.18x10^{-38}$
Maximum exponent (E^{max}) for DOUBLE values	308
Smallest DOUBLE value[125]	$-1.79x10^{308}$
Largest DOUBLE value[125]	$+1.79x10^{308}$
Minimum exponent (E^{min}) for DOUBLE values	-308
Smallest positive DOUBLE value[125]	$+2.23x10^{-308}$
Largest negative DOUBLE value[125]	$-2.23x10^{-308}$
Maximum exponent (E^{max}) for DECFLOAT(16) values	384
Smallest DECFLOAT(16) value[126]	$-9.999999999999999x10^{384}$
Largest DECFLOAT(16) value[126]	$9.999999999999999x10^{384}$
Minimum exponent (E^{min}) for DECFLOAT(16) values	-383
Smallest positive DECFLOAT(16) value[126]	$1x10^{-383}$
Largest negative DECFLOAT(16) value[126]	$-1x10^{-383}$
Maximum exponent (E^{max}) for DECFLOAT(34) values	6144
Smallest DECFLOAT(34) value[126]	$-9.999999999999999999999999999999999x10^{6144}$
Largest DECFLOAT(34) value[126]	$9.999999999999999999999999999999999x10^{6144}$
Minimum exponent (E^{min}) for DECFLOAT(34) values	-6143
Smallest positive DECFLOAT(34) value[126]	$1x10^{-6143}$
Largest negative DECFLOAT(34) value[126]	$-1x10^{-6143}$

125. The values shown are approximate.

126. These are the limits for normal numbers in DECFLOAT. DECFLOAT also contains special values such as NaN and Infinity that are also valid. DECFLOAT also supports subnormal numbers that are outside of the documented range.

Table 113. String Limits

String Limits	DB2 for i Limit
Maximum length of CHAR (in bytes)	32765[127]
Maximum length of VARCHAR (in bytes)	32739[127]
Maximum length of CLOB (in bytes)	2 147 483 647
Maximum length of GRAPHIC (in double-byte characters)	16382[127]
Maximum length of VARGRAPHIC (in double-byte characters)	16369[127]
Maximum length of DBCLOB (in double-byte characters)	1 073 741 823
Maximum length of BINARY (in bytes)	32765[127]
Maximum length of VARBINARY (in bytes)	32739[127]
Maximum length of BLOB (in bytes)	2 147 483 647
Maximum length of serialized XML (in bytes)	2 147 483 647
Maximum length of character constant	32740
Maximum length of a graphic constant	16370
Maximum length of binary constant	32740
Maximum length of concatenated character string	2 147 483 647
Maximum length of concatenated graphic string	1 073 741 823
Maximum length of concatenated binary string	2 147 483 647
Maximum number of hexadecimal constant digits	32 762
Maximum length of catalog comments	2000[128]
Maximum length of column label (in bytes)	60
Maximum length of SQL routine label	128
Maximum length of table, package, or alias label	50
Maximum length of C NUL-terminated	32739[127]
Maximum length of C NUL-terminated graphic	16369[127]

Table 114. XML Limits

XML Limits	DB2 for i Limit
Maximum length of an XML schema document (in bytes)	2 147 483 647
Maximum length of a parsed XML entity	Storage

127. If the column is NOT NULL, the maximum is one more.

128. For sequences the limit is 500.

Table 115. Datetime Limits

Datetime Limits	DB2 for i Limit
Smallest DATE value	0001-01-01
Largest DATE value	9999-12-31
Smallest TIME value	00:00:00
Largest TIME value	24:00:00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000
Largest TIMESTAMP value	9999-12-31-24.00.00.000000

Table 116. DataLink Limits

Datalink Limits	DB2 for i Limit
Maximum length of DATALINK	32718
Maximum length of DATALINK comment	254

Table 117. Database Manager Limits

Database Manager Limits	DB2 for i Limit
Relational Database	
Maximum number of schemas	storage
Maximum number of tables in a relational database	storage
Maximum number of nodes in a nodegroup	32
Schemas	
Maximum number of objects in a schema	approximately 360 000
Tables and Views	
Maximum number of columns in a table	8000
Maximum number of columns in a view	8000
Maximum length of a row without LOBs including all overhead	32766
Maximum length of a row with LOBs including all overhead	3 758 096 383
Maximum number of rows in a non-partitioned table	4 294 967 288
Maximum number of rows in a data partition	4 294 967 288
Maximum size of a non-partitioned table	1.7 terabytes
Maximum size of a data partition	1.7 terabytes
Maximum number of data partitions in a single partitioned table	256
Maximum number of table partitioning columns	120
Maximum number of tables referenced in a view or materialized query table	256[129]
Maximum number of dependent views, materialized query tables, and indexes on a table or view.	storage
Constraints	
Maximum number of constraints on a table	5000
Maximum number of columns in a UNIQUE constraint	120
Maximum combined length of columns in a UNIQUE constraint (in bytes)	32767[127]
Maximum number of referencing columns in a foreign key	120
Maximum combined length of referencing columns in a foreign key (in bytes)	32767[127]
Maximum length of a CHECK constraint (in bytes)	statement
Triggers	
Maximum number of triggers on a table	300
Maximum runtime depth of cascading triggers	200
Indexes	
Maximum number of indexes on a table	approximately 15000
Maximum number of columns in an index key	120
Maximum length of an index key	32767[127]
Maximum size of a non-partitioned index	1.7 terabytes
Maximum size of a partition of a partitioned index	1.7 terabytes
SQL	
Maximum length of an SQL statement (in bytes)	2 097 152
Maximum number of tables referenced in an SQL statement	1000[129]

Table 117. Database Manager Limits (continued)

Database Manager Limits	DB2 for i Limit
Maximum number of variables and constants in an SQL statement	4096[130]
Maximum number of elements in a select list	approximately 8000[131]
Maximum number of predicates in a WHERE or HAVING clause	statement
Maximum number of columns in a GROUP BY clause	total GROUP BY length
Maximum total length of columns in a GROUP BY clause	3.5 Gigabytes[132]
Maximum number of elements in a CUBE grouping	10
Maximum number of columns in an ORDER BY clause	total ORDER BY length
Maximum total length of columns in an ORDER BY clause	3.5 Gigabytes [132]
Maximum levels of recursion for a hierarchical query	250
Maximum levels allowed for a subquery	256
Maximum number of values in an insert operation	8000
Maximum number of SET clauses in a single update operation	8000
Routines	
Maximum number of parameters in a procedure	1024[133]
Maximum number of parameters in a function	1024
Maximum number of nested levels for routines	storage
Types	
Maximum cardinality of an array type	2 147 483 647
Applications	
Maximum number of host variable declarations in a precompiled program	storage[134]
Maximum length of a host variable value (in bytes)	2 147 483 647
Maximum length of an MQ message CLOB value (in bytes)	2M
Maximum length of an MQ message varying length value (in bytes)	32000
Maximum number of declared cursors in a program	storage
Maximum number of cursors opened at one time	storage [135]
Maximum number of rows locked in a unit of work	500 000 000
Maximum number of locators in a transaction	16 000 000 [136]
Maximum size of an SQLDA (in bytes)	16 777 215
Maximum number of prepared statements	storage
Maximum number of savepoints active at one time	storage
Maximum number of simultaneously allocated CLI handles in a process	160 000 [137]
Maximum size of a package	500 megabytes[138]
Maximum length of a path	8843
Maximum number of schemas in a path	268
Maximum length of a password	127
Maximum length of a hint	32
Maximum size of a program, service program, or module associated space (in bytes)	16 777 216
Maximum size of the diagnostics area	90K
Maximum size of an array variable	4GB

Appendix B. Characteristics of SQL statements

This appendix contains information on the characteristics of SQL statements pertaining to the various places where they are used.

- "Actions allowed on SQL statements" on page 1502 shows whether an SQL statement can be executed, prepared interactively or dynamically, and whether the statement is processed by the requester, the server or the precompiler.
- "SQL statement data access classification for routines" on page 1505 shows the level of SQL data access that must be specified to use the SQL statement in a routine.
- "Considerations for using distributed relational database" on page 1508 provides information about the use of SQL statements when the application server is not the same as the application requester.

---

129. The maximum number of partitions (members) referenced may exceed 1000. In CREATE VIEW, DELETE, and UPDATE statements the maximum number of partitions (members) is 256.
130. If the statement is not read-only, the limit is 2048. The limit is approximate and may be less if very large string constants or string variables are used.
131. The limit is based on the size of internal structures generated for the parsed SQL statement.
132. The limit is 32766 if CQE processed the select statement. The limit will be less if an ICU collating sequence or ALWCPYDTA(*NO) is used.
133. SQL procedures are limited to 1024 parameters. The number of parameters for external procedures depends on the PARAMETER STYLE:
    - PARAMETER STYLE GENERAL has a maximum of 1024.
    - PARAMETER STYLE GENERAL WITH NULLS has a maximum of 1023.
    - PARAMETER STYLE SQL or PARAMETER STYLE DB2SQL has a maximum of 508.
    - PARAMETER STYLE JAVA or PARAMETER STYLE DB2GENERAL has a maximum of 90.

    The maximum number of parameters for external procedures is also limited by the maximum number of parameters allowed by the licensed program used to compile the external program.
134. In RPG/400 and PL/I programs when the old parameter passing technique is used, the limit is approximately 4000. The limit is based on the number of pointers allowed in the program. In all other cases, the limit is based on operating system constraints.
135. The maximum number of cursors open at one time in a single job is approximately 21 754.
136. The maximum number of locators in a transaction in SQL Server mode is 209 000.
137. The maximum number of allocated handles per DRDA connection is 500.
138. The maximum size of a DRDA package can be increased to 1 gigabyte by using a QAQQINI option. Other types of packages are limited to 16M.

Actions allowed on SQL statements

Indicates whether an SQL statement can be executed, prepared interactively or dynamically, and whether the statement is processed by the requester, the server, or the precompiler.

The table below shows whether a specific DB2 statement can be executed, prepared interactively or dynamically, or processed by the requester, the server, or the precompiler. The letter **Y** means *yes*.

Table 118. Actions allowed on SQL statements

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
ALLOCATE CURSOR[5]	Y	Y	Y		
ALLOCATE DESCRIPTOR[4] [5]	Y			Y	
ALTER	Y	Y		Y	
ASSOCIATE LOCATORS[5]	Y	Y	Y		
BEGIN DECLARE SECTION[4] [5]					Y
CALL	Y	Y		Y	
CLOSE[4]	Y			Y	
COMMENT	Y	Y		Y	
COMMIT	Y	Y		Y	
CONNECT (Type 1 and Type 2)[4] [5]	Y		Y		
CREATE	Y	Y		Y	
DEALLOCATE DESCRIPTOR[4] [5]	Y			Y	
DECLARE CURSOR[4]					Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y		Y	
DECLARE PROCEDURE[4] [5]					Y
DECLARE STATEMENT[4] [5]					Y
DECLARE VARIABLE[4] [5]					Y
DELETE	Y	Y		Y	
DESCRIBE[4]	Y			Y	
DESCRIBE CURSOR[4] [5]	Y		Y		
DESCRIBE INPUT[4] [5]	Y			Y	
DESCRIBE PROCEDURE[4] [5]	Y		Y		
DESCRIBE TABLE[4]	Y			Y	
DISCONNECT[4] [5]	Y		Y		
DROP	Y	Y		Y	
END DECLARE SECTION[4] [5]					Y
EXECUTE[4]	Y			Y	
EXECUTE IMMEDIATE[4]	Y			Y	
FETCH	Y			Y	
FREE LOCATOR[4] [5]	Y	Y		Y	

Table 118. Actions allowed on SQL statements (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
GET DESCRIPTOR[4] [5]	Y			Y	
GET DIAGNOSTICS[5]	Y			Y	
GRANT	Y	Y		Y	
HOLD LOCATOR[4] [5]	Y	Y		Y	
INCLUDE[4] [5]					Y
INSERT	Y	Y		Y	
LABEL	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
MERGE	Y	Y		Y	
OPEN[4]	Y			Y	
PREPARE[4]	Y			Y	
REFRESH TABLE	Y	Y		Y	
RELEASE connection[4] [5]	Y		Y		
RELEASE SAVEPOINT	Y	Y		Y	
RENAME	Y	Y		Y	
REVOKE	Y	Y		Y	
ROLLBACK	Y	Y		Y	
SAVEPOINT	Y	Y		Y	
SELECT INTO [5]	Y			Y	
SET CONNECTION[4] [5]	Y		Y		
SET CURRENT DEBUG MODE	Y	Y		Y	
SET CURRENT DECFLOAT ROUNDING MODE	Y	Y		Y	
SET CURRENT DEGREE[5]	Y	Y		Y	
SET DESCRIPTOR[4] [5]	Y			Y	
SET ENCRYPTION PASSWORD	Y	Y		Y	
SET CURRENT IMPLICIT XMLPARSE OPTION	Y	Y		Y	
SET OPTION[4] [5]					Y
SET PATH	Y	Y		Y	
SET RESULT SETS[3] [5]	Y			Y	
SET SCHEMA	Y	Y		Y	
SET SESSION AUTHORIZATION[5]	Y	Y		Y	
SET TRANSACTION	Y	Y		Y	
SET transition-variable[1]	Y			Y	
SET variable	Y	Y [6]	Y		
SIGNAL[5]	Y			Y	
SQL-control-statement[2]	Y			Y	

Table 118. Actions allowed on SQL statements (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
UPDATE	**Y**	**Y**		**Y**	
VALUES[1]	**Y**			**Y**	
VALUES INTO [5]	**Y**	**Y**		**Y**	
WHENEVER[4] [5]					**Y**

Notes:

1. This statement can only be used in the triggered action of a trigger.
2. This statement can only be used in an SQL function, SQL procedure, or SQL trigger.
3. This statement can only be used in a procedure.
4. This statement is not applicable in a Java program.
5. This statement is not supported in a REXX program.
6. The target of the SET variable statement must be a global variable.

SQL statement data access classification for routines

Indicates the level of SQL data access that must be specified to use the SQL statement in a routine.

The following table indicates whether an SQL statement (specified in the first column) is allowed to execute in a function or procedure with the specified SQL data access classification. If an executable SQL statement is encountered in a function or procedure defined with NO SQL, SQLSTATE 38001 is returned. For other executions contexts, SQL statements that are not supported in any context return SQLSTATE 38003. For other SQL statements not allowed in a CONTAINS SQL context, SQLSTATE 38004 is returned and in a READS SQL DATA context, SQLSTATE 38002 is returned. During creation of an SQL function or SQL procedure, a statement that does not match the SQL data access classification will cause SQLSTATE 42895 to be returned.

Table 119. SQL Statement and SQL Data Access Classification

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALLOCATE CURSOR			Y	Y
ALLOCATE DESCRIPTOR			Y	Y
ALTER ...				Y
ASSOCIATE LOCATORS			Y	Y
BEGIN DECLARE SECTION	Y[1]	Y	Y	Y
CALL		Y	Y	Y
CLOSE			Y	Y
COMMENT				Y
COMMIT[3]		Y	Y	Y
CONNECT (Type 1 and Type 2)[3]				
CREATE ...				Y
DEALLOCATE DESCRIPTOR			Y	Y
DECLARE CURSOR	Y[1]	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE				Y
DECLARE PROCEDURE	Y[1]	Y	Y	Y
DECLARE STATEMENT	Y[1]	Y	Y	Y
DECLARE VARIABLE	Y[1]	Y	Y	Y
DELETE				Y
DESCRIBE			Y	Y
DESCRIBE CURSOR			Y	Y
DESCRIBE INPUT			Y	Y
DESCRIBE PROCEDURE			Y	Y
DESCRIBE TABLE			Y	Y
DISCONNECT[3]				
DROP ...				Y

Table 119. SQL Statement and SQL Data Access Classification (continued)

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
END DECLARE SECTION	Y[1]	Y	Y	Y
EXECUTE		Y[2]	Y[2]	Y
EXECUTE IMMEDIATE		Y[2]	Y[2]	Y
FETCH			Y	Y
FREE LOCATOR		Y	Y	Y
GET DESCRIPTOR			Y	Y
GET DIAGNOSTICS		Y	Y	Y
GRANT ...				Y
HOLD LOCATOR		Y	Y	Y
INCLUDE	Y[1]	Y	Y	Y
INSERT				Y
LABEL				Y
LOCK TABLE		Y	Y	Y
MERGE				Y
OPEN			Y	Y
PREPARE		Y	Y	Y
REFRESH TABLE				Y
RELEASE CONNECTION[3]				
RELEASE SAVEPOINT				Y
RENAME				Y
REVOKE ...				Y
ROLLBACK[3]		Y	Y	Y
ROLLBACK TO SAVEPOINT				Y
SAVEPOINT				Y
SELECT INTO			Y	Y
SET CONNECTION[3]				
SET CURRENT DEBUG MODE			Y	Y
SET CURRENT DECFLOAT ROUNDING MODE		Y	Y	Y
SET CURRENT DEGREE			Y	Y
SET CURRENT IMPLICIT XMLPARSE OPTION		Y	Y	Y
SET DESCRIPTOR			Y	Y
SET ENCRYPTION PASSWORD		Y	Y	Y
SET OPTION	Y[1]	Y	Y	Y
SET PATH		Y	Y	Y
SET RESULT SETS		Y	Y	Y
SET SCHEMA			Y	Y

Table 119. SQL Statement and SQL Data Access Classification (continued)

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
SET SESSION AUTHORIZATION			Y	Y
SET TRANSACTION		Y	Y	Y
SET variable		Y	Y	Y
SIGNAL		Y	Y	Y
UPDATE				Y
VALUES				
VALUES INTO			Y	Y
WHENEVER	Y[1]	Y	Y	Y

Note:

1. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
2. It depends on the statement being executed. The statement specified for the EXECUTE statement must be a statement that is allowed in the context of the particular SQL access level in effect. For example, if the SQL access level in effect is READS SQL DATA, the statement must not be an INSERT, UPDATE, or DELETE.
3. Connection management and transaction statements are not allowed in a procedure running on a remote server. COMMIT and ROLLBACK are not allowed in an ATOMIC SQL procedure.

Considerations for using distributed relational database

This section contains information that may be useful in developing applications that use application servers which are not the same product as their application requesters.

All DB2 products support extensions to IBM SQL. Some of these extensions are product-specific, but many are already shared by more than one product or support is planned but not yet generally available.

For the most part, an application can use the statements and clauses that are supported by the database manager of the current server, even though that application might be running through the application requester of a database manager that does not support some of those statements and clauses. Restrictions to this general rule are identified by application requester:

- for DB2 for z/OS Application Server application requester, see Table 120
- for DB2 for i Application Server application requester, seeTable 121 on page 1509
- for DB2 LUW application requester, see Table 122 on page 1509.

Note that an 'R' in the table indicates that this SQL function is not supported in the specified environment. An 'R' in every column of the same row means that the function is available only if the current server and requester are the same product or that the statement is blocked by the application requester from being processed at the application server.

Table 120. DB2 for z/OS Application Requester

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 LUW Application Server
COMMIT HOLD	R	R	R
DECLARE STATEMENT			
DECLARE TABLE			
DECLARE VARIABLE			
DESCRIBE TABLE			R
DESCRIBE with USING clause			R
DISCONNECT	R	R	R
ROWID data types			R
DATALINK data types	R	R	R
BINARY and VARBINARY data types			R
Host declarations not documented in language specific appendices		[139]	[139]
PREPARE with USING clause			R
ROLLBACK HOLD	R	R	R
SET CURRENT PACKAGESET			
SET variable		R	R
SET TRANSACTION	R	R	R
Scrollable Cursor statements	R	R	R
UPDATE cursor - FOR UPDATE clause not specified			

Table 121. DB2 for i Application Requester

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 LUW Application Server
COMMIT HOLD	R		R
DECLARE STATEMENT			
DECLARE TABLE			
DECLARE VARIABLE			
DESCRIBE TABLE			R
DESCRIBE with USING clause			R
DISCONNECT			
Host Variables - optional colon	R	R	R
ROWID data types			R
DATALINK data types	R		R
BINARY and VARBINARY data types			R
Host declarations not documented in language specific appendices	[139]		[139]
PREPARE with USING clause			R
ROLLBACK HOLD	R		R
SET CURRENT PACKAGESET	R	R	R
SET variable	R	R	R
SET TRANSACTION	R		R
Scrollable Cursor statements	R		R
UPDATE cursor - FOR UPDATE clause not specified	R		

Table 122. DB2 LUW application requester

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 LUW Application Server
COMMIT HOLD	R	R	R
DECLARE STATEMENT	R	R	R
DECLARE TABLE	R	R	R
DECLARE VARIABLE	R	R	R
DESCRIBE TABLE	R	R	R
DESCRIBE with USING clause	R	R	R
DISCONNECT			
Host Variables - optional colon	R	R	R
ROWID data types	[140]	[140]	R
DATALINK data types	R	R	R
BINARY and VARBINARY data types	R	R	R
Host declarations not documented in language specific appendices	[139]	[139]	
PREPARE with USING clause	R	R	R

139. The statement is supported if the application requester understands it.

Table 122. DB2 LUW application requester (continued)

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 LUW Application Server
ROLLBACK HOLD	R	R	R
SET CURRENT PACKAGESET			
SET variable	R	R	R
SET TRANSACTION	R	R	R
Scrollable Cursor statements	R	R	R
UPDATE cursor - FOR UPDATE clause not specified	R		

140. The DB2 LUW application requester application requester will process a ROWID data type at the application server using the compatible VARCHAR(40) FOR BIT DATA data type.

CONNECT (Type 1) and CONNECT (Type 2) differences

There are two types of CONNECT statements.

They have the same syntax, but they have different semantics:

- CONNECT (Type 1) is used for remote unit of work. See "Remote unit of work" on page 42.
- CONNECT (Type 2) is used for distributed unit of work. See "CONNECT (Type 2)" on page 842.

The following table summarizes the differences between CONNECT (Type 1) and CONNECT (Type 2) rules:

Table 123. CONNECT (Type 1) and CONNECT (Type 2) Differences

Type 1 Rules	Type 2 Rules
CONNECT statements can only be executed when the activation group is in the connectable state. No more than one CONNECT statement can be executed within the same unit of work.	More than one CONNECT statement can be executed within the same unit of work. There are no rules about the connectable state.
If the CONNECT statement fails because the server name is not listed in the local directory, the connection state of the activation group is unchanged. If a CONNECT statement fails because the activation group is not in the connectable state, the SQL connection status of the activation group is unchanged. If a CONNECT statement fails for any other reason, the activation group is placed in the unconnected state.	If a CONNECT statement fails, the current SQL connection is unchanged and any subsequent SQL statements are executed by the current server.
CONNECT ends all existing connections of the activation group. Accordingly, CONNECT also closes any open cursors for that activation group.	CONNECT does not end connections and does not close cursors.
A CONNECT to the current server is executed like any other CONNECT (Type 1) statement.	A CONNECT to the current server causes an error.

Determining the CONNECT rules that apply

A program preparation option is used to specify the type of CONNECT that will be performed by a program. The program preparation option is specified using the RDBCNNMTH parameter on the CRTSQLxxx command.

Connecting to servers that only support remote unit of work

CONNECT (Type 2) connections to application servers that only support remote unit of work might result in connections that are read-only.

If a CONNECT (Type 2) is performed to an application server that only supports remote unit of work:

- The connection allows read-only operations if, at the time of the connect, there are any dormant connections that allow updates. In this case, the connection does not allow updates.
- Otherwise, the connection allows updates.

If a CONNECT (Type 2) is performed to an application server that supports distributed unit of work:

- The connection allows read-only operations when there are dormant connections that allow updates to application servers that only support remote unit of work. In this case, the connection allows updates as soon as the dormant connection is ended.
- Otherwise, the connection allows updates.

141. DB2 for i using the initial DRDA support for native TCP/IP is an example of an application server that supports only remote unit of work.

Appendix C. SQLCA (SQL communication area)

An SQLCA is a set of variables that may be updated at the end of the execution of every SQL statement. A program that contains executable SQL statements may provide one, but no more than one SQLCA (unless a stand-alone SQLCODE or a stand-alone SQLSTATE variable is used instead), except in Java, where the SQLCA is not applicable.

Instead of using an SQLCA, the GET DIAGNOSTICS statement can be used in all languages to return return codes and other information about the previous SQL statement. For more information, see "GET DIAGNOSTICS" on page 1194.

The SQL INCLUDE statement can be used to provide the declaration of the SQLCA in all host languages except Java, RPG, or REXX. For information about the use of the SQLCA in a REXX procedure, see the Embedded SQL Programming topic collection. For information about how to access the information regarding errors and warnings in Java, refer to the IBM Developer Kit for Java topic collection.

In C, COBOL, and PL/I, the name of the storage area must be SQLCA. Every SQL statement must be within the scope of its declaration.

If a stand-alone SQLCODE or SQLSTATE is specified in the program, the SQLCA must not be included. For more information, see "SQL diagnostic information" on page 708.

The stand-alone SQLCODE and stand-alone SQLSTATE must not be specified in the Java or REXX language.

Field descriptions

The names in the following table are those provided by the SQL INCLUDE statement.

For the most part, C (and C++), COBOL, and PL/I use the same names. RPG names are different, because in RPG/400, they are limited to 6 characters. In ILE RPG, both a long name and the short 6 character name is supported. Note one instance where PL/I names differ from the COBOL names.

Table 124. Names Provided by the SQL INCLUDE Statement

C Name COBOL Name PL/I Name	ILE RPG Name RPG/400 Name	Field Data Type	Field Value
SQLCAID sqlcaid	SQLCAID SQLAID	CHAR(8)	An "eye catcher" for storage dumps, containing 'SQLCA'.
SQLCABC sqlcabc	SQLCABC SQLABC	INTEGER	Contains the length of the SQLCA, 136.

Table 124. Names Provided by the SQL INCLUDE Statement (continued)

C Name COBOL Name PL/I Name	ILE RPG Name RPG/400 Name	Field Data Type	Field Value
SQLCODE sqlcode	SQLCODE SQLCOD	INTEGER	Contains an SQL return code. **Code** **Meaning** **0** Successful execution although SQLWARN indicators might have been set. **positive** Successful execution, but with a warning condition. **negative** Error condition.
SQLERRML[1] sqlerrml	SQLERRML SQLERL	SMALLINT	Length indicator for SQLERRMC, in the range 0 through 70. 0 means that the value of SQLERRMC is not pertinent.
SQLERRMC[1] sqlerrmc	SQLERRMC SQLERM	CHAR(70)	Contains message replacement text associated with the SQLCODE. For CONNECT and SET CONNECTION, the SQLERRMC field contains information about the connection, see Table 127 on page 1517 for a description of the replacement text.
SQLERRP sqlerrp	SQLERRP SQLERP	CHAR(8)	Contains the name of the product and module returning the error or warning. The first three characters identify the product: • ARI for DB2 for VM and VSE • DSN for DB2 for z/OS • QSQ for DB2 for i • SQL for all other DB2 products See "CONNECT (Type 1)" on page 836 or "CONNECT (Type 2)" on page 842 for additional information.
SQLERRD sqlerrd	SQLERRD SQLERR[2]	Array	Contains six INTEGER variables that provide diagnostic information, see Table 126 on page 1515 for a description of the diagnostic information.
SQLWARN sqlwarn	SQLWARN SQLWRN[3]	CHAR(11)	A set of 11 CHAR(1) warning indicators, each containing blank or 'W' or 'N'.
SQLSTATE sqlstate	SQLSTATE SQLSTT	CHAR(5)	A return code that indicates the outcome of the most recently executed SQL statement.

Notes:

1 In COBOL, SQLERRM includes SQLERRML and SQLERRMC. In PL/I, the varying-length string SQLERRM is equivalent to SQLERRML prefixed to SQLERRMC.

2 In RPG/400, SQLERR is defined as 24 characters (not an array) that are redefined by the fields SQLER1 through SQLER6. The fields are full-word binary. In ILE RPG, SQLERR is also redefined as an array. The name of the array is SQLERRD.

3 In RPG/400, SQLWRN is defined as 11 characters (not an array) that are redefined by the fields SQLWN0 through SQLWNA. The fields are full-word binary. In ILE RPG, SQLWRN is also redefined as an array. The name of the array is SQLWARN.

Table 125. SQLWARN Diagnostic Information

C Name COBOL Name PL/I Name	ILE RPG Name RPG/400 Name	Field Value
SQLWARN0 sqlwarn[0]	SQLWARN(1) SQLWN0	Contains 'W' if at least one other indicator contains 'W' or 'N', it is blank if all other indicators are blank.
SQLWARN1 sqlwarn[1]	SQLWARN(2) SQLWN1	Contains 'W' if the value of a string column was truncated when assigned to a host variable. Contains 'N' if *NOCNULRQD was specified an the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement) and if the value of a string column was assigned to a C NUL-terminated host variable and if the host variable was large enough to contain the result but not large enough to contain the NUL-terminator.
SQLWARN2 sqlwarn[2]	SQLWARN(3) SQLWN2	Contains 'W' if the null values were eliminated from the argument of a function; not necessarily set to 'W' for the MIN or MAX function because its results are not dependent on the elimination of null values.
SQLWARN3 sqlwarn[3]	SQLWARN(4) SQLWN3	Contains 'W' if the number of columns is larger than the number of host variables.
SQLWARN4 sqlwarn[4]	SQLWARN(5) SQLWN4	Contains 'W' if a prepared UPDATE or DELETE statement does not include a WHERE clause.
SQLWARN5 sqlwarn[5]	SQLWARN(6) SQLWN5	Contains a character value of 1 (read only), 2 (read and delete), or 4 (read, delete, and update) to reflect capability of a cursor after the OPEN statement.
SQLWARN6 sqlwarn[6]	SQLWARN(7) SQLWN6	Contains 'W' if date arithmetic results in an end-of-month adjustment.
SQLWARN7 sqlwarn[7]	SQLWARN(8) SQLWN7	Reserved
SQLWARN8 sqlwarn[8]	SQLWARN(9) SQLWN8	Contains 'W' if the result of a character conversion contains the substitution character.
SQLWARN9 sqlwarn[9]	SQLWARN(10) SQLWN9	Reserved
SQLWARNA sqlwarn[10]	SQLWARN(11) SQLWNA	Reserved

Table 126. SQLERRD Diagnostic Information

C Name COBOL Name PL/I Name	ILE RPG Name RPG/400 Name	Field Value
SQLERRD(1) sqlerrd[0]	SQLERRD(1) SQLER1	Contains the last four characters of the CPF escape message if SQLCODE is less than 0. For example, if the message is CPF5715, X'F5F7F1F5' is placed in SQLERRD(1).[1] For a call to a procedure, contains the return status value specified on the RETURN statement. If a return status value is not specified on the RETURN statement or the procedure is an external procedure, • 0 is returned if the CALL statement is successful, or • –1 is returned if the CALL statement is not successful.

Table 126. SQLERRD Diagnostic Information (continued)

C Name COBOL Name PL/I Name	ILE RPG Name RPG/400 Name	Field Value
SQLERRD(2) sqlerrd[1]	SQLERRD(2) SQLER2	Contains the last four characters of a CPD diagnostic message if the SQL code is less than 0.[1] For a CALL statement, SQLERRD(2) contains the number of result sets. For an OPEN statement, if the cursor is insensitive to changes, SQLERRD(2) contains the actual number of rows in the result set. If the cursor is sensitive to changes, SQLERRD(2) contains an estimated number of rows in the result set.
SQLERRD(3) sqlerrd[2]	SQLERRD(3) SQLER3	For a CONNECT for status statement, SQLERRD(3) contains information about the connection status. See "CONNECT (Type 2)" on page 842 for more information. For INSERT, MERGE, UPDATE, REFRESH, and DELETE, shows the number of rows affected. For a FETCH statement, SQLERRD(3) contains the number of rows fetched. For the PREPARE statement, contains the estimated number of rows selected. If the number of rows is greater than 2 147 483 647, then 2 147 483 647 is returned.
SQLERRD(4) sqlerrd[3]	SQLERRD(4) SQLER4	For the PREPARE statement, contains a relative number estimate of the resources required for every execution. This number varies depending on the current availability of indexes, file sizes, CPU model, etc. It is an estimated cost for the access plan chosen by the DB2 for i Query Optimizer. For a CONNECT and SET CONNECTION statement, SQLERRD(4) contains the type of conversation used and whether or not committable updates can be performed. See "CONNECT (Type 2)" on page 842 for more information. For a CALL statement, SQLERRD(4) contains the message key of the error that caused the procedure to fail. The QMHRTVPM API can be used to return the message description for the message key. For a trigger error in a DELETE, INSERT MERGE, or UPDATE statement, SQLERRD(4) contains the message key of the error that was signaled from the trigger program. The QMHRTVPM API can be used to return the message description for the message key. For a FETCH statement, if the result row does not contain a LOB, SQLERRD(4) contains the length of the row(s) retrieved. For an OPEN statement, if the result row does not contain a LOB, SQLERRD(4) contains the length of a result row.

Table 126. SQLERRD Diagnostic Information (continued)

C Name COBOL Name PL/I Name	ILE RPG Name RPG/400 Name	Field Value
SQLERRD(5) sqlerrd[4]	SQLERRD(5) SQLER5	For a CONNECT or SET CONNECTION statement, SQLERRD(5) contains: • -1 if the connection is unconnected • 0 if the connection is local • 1 if the connection is remote For a DELETE, INSERT, MERGE, or UPDATE statement, shows the number of rows affected by referential constraints and triggers. For an EXECUTE IMMEDIATE or PREPARE statement, may contain the position of a syntax error. For a multiple-row FETCH statement, SQLERRD(5) contains +100 if the last row currently in the table has been fetched. For a PREPARE statement, SQLERRD(5) contains the number of parameter markers in the prepared statement.
SQLERRD(6) sqlerrd[5]	SQLERRD(6) SQLER6	Contains the SQL completion message identifier when the SQLCODE is 0. In all other cases, it is undefined.

Note:

[1] SQLERRD(1) and SQLERRD(2) are set only if appropriate and only if the current server is DB2 for i.

Table 127. SQLERRMC Replacement Text for CONNECT and SET CONNECTION

Description	Data type
Relational Database Name	CHAR(18)
Product Identification (same as SQLERRP)	CHAR(8)
User ID of the server job	CHAR(10)
Connection method (*DUW or *RUW)	CHAR(10)

Table 127. SQLERRMC Replacement Text for CONNECT and SET CONNECTION (continued)

Description	Data type
DDM server class name	CHAR(10)
QAS DB2 for i	
QDB2 DB2 for z/OS	
QDB2/6000 DB2 for AIX	
QDB2/HPUX DB2 for HP-UX**	
QDB2/LINUX DB2 for Linux	
QDB2/NT DB2 for Windows** NT, 2000, and XP	
QDB2/SUN DB2 for SUN** Solaris**	
QSQLDS/VM DB2 Server for VM	
QSQLDS/VSE DB2 Server for VSE	
Connection type (same as SQLERRD(4))	SMALLINT

INCLUDE SQLCA declarations

This section shows the equivalent INCLUDE SQLCA declaration for C and C++, COBOL, PL/I, RPG/400, and ILE RPG.

In **C and C++**, INCLUDE SQLCA declarations are equivalent to the following:

```
#ifndef    SQLCODE
struct     sqlca
{
           unsigned char  sqlcaid[8];
           long           sqlcabc;
           long           sqlcode;
           short          sqlerrml;
           unsigned char  sqlerrmc[70];
           unsigned char  sqlerrp[8];
           long           sqlerrd[6];
           unsigned char  sqlwarn[11];
           unsigned char  sqlstate[5];
};
#define      SQLCODE   sqlca.sqlcode
#define      SQLWARN0  sqlca.sqlwarn[0]
#define      SQLWARN1  sqlca.sqlwarn[1]
#define      SQLWARN2  sqlca.sqlwarn[2]
#define      SQLWARN3  sqlca.sqlwarn[3]
#define      SQLWARN4  sqlca.sqlwarn[4]
#define      SQLWARN5  sqlca.sqlwarn[5]
#define      SQLWARN6  sqlca.sqlwarn[6]
#define      SQLWARN7  sqlca.sqlwarn[7]
#define      SQLWARN8  sqlca.sqlwarn[8]
#define      SQLWARN9  sqlca.sqlwarn[9]
#define      SQLWARNA  sqlca.sqlwarn[10]
#define      SQLSTATE  sqlca.sqlstate
#endif
struct sqlca sqlca;
```

In **COBOL**, INCLUDE SQLCA declarations are equivalent to the following:

```
01 SQLCA.
   05 SQLCAID      PIC X(8).
   05 SQLCABC      PIC S9(9) BINARY.
   05 SQLCODE      PIC S9(9) BINARY.
   05 SQLERRM.
      49 SQLERRML  PIC S9(4) BINARY.
      49 SQLERRMC  PIC X(70).
   05 SQLERRP      PIC X(8).
   05 SQLERRD      OCCURS 6 TIMES
                   PIC S9(9) BINARY.
   05 SQLWARN.
      10 SQLWARN0  PIC X(1).
      10 SQLWARN1  PIC X(1).
      10 SQLWARN2  PIC X(1).
      10 SQLWARN3  PIC X(1).
      10 SQLWARN4  PIC X(1).
      10 SQLWARN5  PIC X(1).
      10 SQLWARN6  PIC X(1).
      10 SQLWARN7  PIC X(1).
      10 SQLWARN8  PIC X(1).
      10 SQLWARN9  PIC X(1).
      10 SQLWARNA  PIC X(1).
   05 SQLSTATE     PIC X(5).
```

Note: In COBOL, INCLUDE SQLCA must not be specified outside the Working Storage Section.

In **PL/I**; INCLUDE SQLCA declarations are equivalent to the following:

```
DCL 1 SQLCA,
      2 SQLCAID      CHAR(8),
      2 SQLCABC      BIN FIXED(31),
      2 SQLCODE      BIN FIXED(31),
      2 SQLERRM      CHAR(70) VAR,
      2 SQLERRP      CHAR(8),
      2 SQLERRD(6)   BIN FIXED(31),
      2 SQLWARN,
        3 SQLWARN0   CHAR(1),
        3 SQLWARN1   CHAR(1),
        3 SQLWARN2   CHAR(1),
        3 SQLWARN3   CHAR(1),
        3 SQLWARN4   CHAR(1),
        3 SQLWARN5   CHAR(1),
        3 SQLWARN6   CHAR(1),
        3 SQLWARN7   CHAR(1),
        3 SQLWARN8   CHAR(1),
        3 SQLWARN9   CHAR(1),
        3 SQLWARNA   CHAR(1),
      2 SQLSTATE     CHAR(5);
```

In **RPG/400**; SQLCA declarations are equivalent to the following:

```
ISQLCA          DS
I                                         1   8 SQLAID            SQL
I                                    B    9  120SQLABC            SQL
I                                    B   13  160SQLCOD            SQL
I                                    B   17  180SQLERL            SQL
I                                        19  88 SQLERM            SQL
I                                        89  96 SQLERP            SQL
I                                        97 120 SQLERR            SQL
I                                    B   97 1000SQLER1            SQL
I                                    B  101 1040SQLER2            SQL
I                                    B  105 1080SQLER3            SQL
I                                    B  109 1120SQLER4            SQL
I                                    B  113 1160SQLER5            SQL
I                                    B  117 1200SQLER6            SQL
I                                      121 131 SQLWRN            SQL
I                                      121 121 SQLWN0            SQL
I                                      122 122 SQLWN1            SQL
I                                      123 123 SQLWN2            SQL
I                                      124 124 SQLWN3            SQL
I                                      125 125 SQLWN4            SQL
I                                      126 126 SQLWN5            SQL
I                                      127 127 SQLWN6            SQL
I                                      128 128 SQLWN7            SQL
I                                      129 129 SQLWN8            SQL
I                                      130 130 SQLWN9            SQL
I                                      131 131 SQLWNA            SQL
I                                      132 136 SQLSTT            SQL
```

In **ILE RPG**; SQLCA declarations are equivalent to the following:

```
 D*      SQL Communications area
 D SQLCA           DS
 D  SQLCAID                       8A   INZ(X'0000000000000000')
 D  SQLAID                        8A   OVERLAY(SQLCAID)
 D  SQLCABC                      10I 0
 D  SQLABC                        9B 0 OVERLAY(SQLCABC)
 D  SQLCODE                      10I 0
 D  SQLCOD                        9B 0 OVERLAY(SQLCODE)
 D  SQLERRML                      5I 0
 D  SQLERL                        4B 0 OVERLAY(SQLERRML)
 D  SQLERRMC                     70A
 D  SQLERM                       70A   OVERLAY(SQLERRMC)
 D  SQLERRP                       8A
 D  SQLERP                        8A   OVERLAY(SQLERRP)
 D  SQLERR                       24A
```

```
D   SQLER1                      9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER2                      9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER3                      9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER4                      9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER5                      9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER6                      9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLERRD                    10I 0 DIM(6)  OVERLAY(SQLERR)
D  SQLWRN                      11A
D   SQLWN0                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN1                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN2                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN3                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN4                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN5                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN6                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN7                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN8                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN9                      1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWNA                      1A   OVERLAY(SQLWRN:*NEXT)
D  SQLWARN                      1A   DIM(11) OVERLAY(SQLWRN)
D  SQLSTATE                     5A
D  SQLSTT                       5A   OVERLAY(SQLSTATE)
D*  End of SQLCA
```

Appendix D. SQLDA (SQL descriptor area)

An SQLDA is a set of variables that is used for execution of the SQL DESCRIBE statement, and it may optionally be used by the PREPARE, OPEN, CALL, FETCH, and EXECUTE statements.

An SQLDA can be used in a DESCRIBE or PREPARE statement, altered with the addresses of storage areas[142], and then used again in a FETCH statement.

SQLDAs are supported for all languages, but predefined declarations are provided only for C (and C++), COBOL, ILE RPG, PL/I, and REXX. In REXX, the SQLDA is somewhat different than in the other languages; for information about the use of SQLDAs in REXX, see the Embedded SQL Programming topic collection.

The meaning of the information in an SQLDA depends on its use.

- When an SQLDA is used in a DESCRIBE or PREPARE statement, an SQLDA provides information to an application program about a prepared *select-statement*. Each column of the result table is described in an SQLVAR occurrence or set of related SQLVAR occurrences.
- In OPEN, EXECUTE, CALL, and FETCH, an SQLDA provides information to the database manager about storage areas for input or output data. Each storage area is described in the SQLVARs.
  - For OPEN and EXECUTE of a statement other than CALL, each SQLVAR occurrence or set of related SQLVAR occurrences describes a storage area that is used to contain an input value which is substituted for a parameter marker in the associated SQL statement that was previously prepared.
  - For FETCH, each SQLVAR occurrence or set of related SQLVAR occurrences describes a storage area that is used to contain an output value from a row of the result table.
  - For CALL and EXECUTE of a prepared CALL statement, each SQLVAR occurrence or set of related SQLVAR occurrences describes a storage area that is used to contain an input or output value (or both) that corresponds to an argument in the argument list for the procedure.

An SQLDA consists of four variables in a header followed by an arbitrary number of occurrences of a *base SQLVAR*. When the SQLDA desribes either LOBs or distint types the base SQLVARs are followed by the same number of occurrences of an *extended SQLVAR*.

Base SQLVAR entry
The base SQLVAR entry is always present. The fields of this entry contain the base information about the column or variable such as data type code, length attribute (except for LOBs), column name (or label), CCSID, variable address, and indicator variable address.

Extended SQLVAR entry
The extended SQLVAR entry is needed (for each column) if the result includes any LOB or distinct type columns. For distinct types, the extended SQLVAR contains the distinct type name. For LOBs, the extended SQLVAR contains the length attribute of the variable and a pointer to the buffer that

142. A storage area could be the storage for a variable defined in the program (that may also be a host variable) or an area of storage explicitly allocated by the application.

contains the actual length. If locators or file reference variables are used to represent LOBs, an extended SQLVAR is not necessary.

The extended SQLVAR entry is also needed for each column when:

- USING BOTH is specified, which indicates that column names and labels are returned.
- USING ALL is specified, which indicates that column names, labels, and system column names are returned.

The fields in the extended SQLVAR that return LOB and distinct type information do not overlap, and the fields that return LOB and label information do not overlap. Depending on the combination of labels, LOBs and distinct types, more than one extended SQLVAR entry per column may be required to return the information. See "Determining how many SQLVAR occurrences are needed" on page 1526.

Field descriptions in an SQLDA header

An SQLDA consists of four variables in a header structure followed by an arbitrary number of occurrences of a sequence of five variables collectively named SQLVAR. In OPEN, CALL, FETCH, and EXECUTE, each occurrence of SQLVAR describes a variable. In PREPARE and DESCRIBE, each occurrence describes a column of a result table.

The SQL INCLUDE statement provides the following field names:

Table 128. Field Descriptions for an SQLDA Header

C Name [143] PL/I Name COBOL Name	Field Data Type	Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the user prior to executing the statement)
sqldaid SQLDAID	CHAR(8)	An 'eye catcher' for storage dumps, containing 'SQLDA '. The 7th byte of the SQLDAID can be used to determine whether more than one SQLVAR entry is needed for each column. For details, see "Determining how many SQLVAR occurrences are needed" on page 1526.	A '2' in the 7th byte indicates that two SQLVAR entries were allocated for each column. A '3' in the 7th byte indicates that three SQLVAR entries were allocated for each column. A '4' in the 7th byte indicates that four SQLVAR entries were allocated for each column.
sqldabc SQLDABC	INTEGER	Length of the SQLDA.	Number of bytes of storage allocated for the SQLDA. Enough storage must be allocated to contain SQLN occurrences. SQLDABC must be set to a value greater than or equal to 16+SQLN*(80), where 80 is the length of an SQLVAR occurrence. If LOBs or distinct types are specified, there must be two SQLVAR entries for each parameter marker.
sqln SQLN	SMALLINT	Unchanged by the database manager. Must be set to a value greater than or equal to zero before the PREPARE or DESCRIBE statement is executed. It should be set to a value that is greater than or equal to the number of columns in the result or a multiple of the number of columns in the result when multiple sets of SQLVAR entries are necessary. Indicates the total number of occurrences of SQLVAR.	Total number of occurrences of SQLVAR provided in the SQLDA. SQLN must be set to a value greater than or equal to zero. If LOBs or distinct types are specified, there must be two SQLVAR entries for each parameter marker and SQLN must be set to two times the number of parameter markers.
sqld SQLD	SMALLINT	The number of columns described by occurrences of SQLVAR (zero if the statement being described is not a select-statement, CALL, or VALUES INTO).	Number of occurrences of SQLVAR entries in the SQLDA that are used when executing the statement. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

143. In this column, the lowercase name is the C Name. The uppercase name is the COBOL, PL/I, or RPG Name.

Determining how many SQLVAR occurrences are needed

The number of SQLVAR occurrences needed depends on the statement that the SQLDA was provided for and the data types of the columns or parameters being described. See the tables above for more information.

The 7th byte of SQLDAID is always set to the number of sets of SQLVARs necessary when LOBs or UDTs are in the result set.

If SQLD is not set to a sufficient number of SQLVAR occurrences:

- When LOBs and UDTs are not in the result set, SQLD is set to the total number of SQLVAR occurrences needed for all sets. When there are LOBs or UDTs in the result set, SQLD is set to the number of columns in the result table and the seventh byte of SQLDAID indicates the number of sets of SQLVAR entries needed. The number of required SQLVAR entries can always be determined by multiplying SQLD by the value in the seventh byte of SQLDAID.
- A warning (SQLSTATE 01594) is returned if at least enough SQLVARs were specified for the Base SQLVAR Entries. The Base SQLVAR entries are returned, but no extended SQLVARs are returned.
- A warning (SQLSTATE 01005) is returned if enough SQLVARs were not specified for even the Base SQLVAR Entries. No SQLVAR entries are returned.[144]

Table 129, Table 130 on page 1527, and Table 131 on page 1527 show how to map the base and extended SQLVAR entries. For an SQLDA that contains both base and extended SQLVAR entries, the base SQLVAR entries are in the first block, followed by a block of extended SQLVAR entries, which if necessary, are followed by a second or third block of extended SQLVAR entries. In each block, the number of occurrences of the SQLVAR entry is equal to the value in SQLD even though many of the extended SQLVAR entries might be unused.

Table 129. Contents of SQLVAR Arrays for USING NAMES, USING SYSTEM NAMES, USING LABELS or USING ANY

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)	Third Set (Extended)	Fourth Set (Extended)
No	No	Blank	n	Column names, system column names, or labels	Not used	Not used	Not used
Yes	No	2	2n	Column names, system column names, or labels	LOBs	Not used	Not used
No	Yes	2	2n	Column names, system column names, or labels	Distinct types	Not used	Not used

144. If LOBs or UDTs are not in the result set, the warning is only returned if the standards option is specified. For information about the standards option, see "Standards compliance" on page xi.

Table 129. Contents of SQLVAR Arrays for USING NAMES, USING SYSTEM NAMES, USING LABELS or USING ANY (continued)

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)	Third Set (Extended)	Fourth Set (Extended)
Yes	Yes	2	2n	Column names, system column names, or labels	LOBs and distinct types	Not used	Not used

Table 130. Contents of SQLVAR Arrays for USING BOTH

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)	Third Set (Extended)	Fourth Set (Extended)
No	No	2	2n	Column names	Labels	Not used	Not used
Yes	No	2	2n	Column names	LOBs and labels	Not used	Not used
No	Yes	3	3n	Column names	Distinct types	Labels	Not used
Yes	Yes	3	3n	Column names	LOBs and distinct types	Labels	Not used

Table 131. Contents of SQLVAR Arrays for USING ALL

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)	Third Set (Extended)	Fourth Set (Extended)
No	No	3	3n	System column names	Labels	Column names	Not used
Yes	No	3	3n	System column names	LOBs and labels	Column names	Not used
No	Yes	4	4n	System column names	Distinct types	Labels	Column names
Yes	Yes	4	4n	System column names	LOBs and distinct types	Labels	Column names

Field descriptions in an occurrence of SQLVAR

This section includes field descriptions in an occurrence of a base and secondary SQLVAR.

Fields in an occurrence of a base SQLVAR

Table 132. Field Descriptions for an SQLVAR

C Name [145] COBOL Name PL/I Name RPG Name	Field Data Type	Usage in DESCRIBE and PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, and EXECUTE (set by the user prior to executing the statement)
sqltype SQLTYPE	SMALLINT	Indicates the data type of the column and whether it can contain nulls. For a description of the type codes, see Table 134 on page 1532. For a distinct type, the data type on which the distinct type is based is placed in this field. The base SQLVAR contains no indication that this is part of the description of a distinct type.	Indicates the data type of the host variable and whether an indicator variable is provided. For a description of the type codes, see Table 134 on page 1532.
sqllen SQLLEN	SMALLINT	The length attribute of the column. For datetime columns, the length of the string representation of the values. See Table 134 on page 1532. For a LOB, the value is 0 regardless of the length attribute of the LOB. For XML, the value is 0. Field SQLLONGLEN in the extended SQLVAR entry contains the length attribute of the LOB or XML.	The length attribute of the host variable. See Table 134 on page 1532. For a LOB, the value is 0 regardless of the length attribute of the LOB. Field SQLLONGLEN in the extended SQLVAR entry contains the length attribute of the LOB. For XML AS BLOB, CLOB, or DBCLOB, the value is 0.
sqlres SQLRES	CHAR(12)	Reserved. Provides boundary alignment for SQLDATA.	Reserved. Provides boundary alignment for SQLDATA.
sqldata SQLDATA	pointer	The CCSID of a string column or XML column as described in Table 135 on page 1534.	Contains the address of the host variable. For LOB host variables, if the SQLDATALEN field in the extended SQLVAR is null, this points to the four-byte LOB length, followed immediately by the LOB data. If the SQLDATALEN field in the extended SQLVAR is not null, this points to the LOB data and the SQLDATALEN field points to the four-byte LOB length.

145. In this column, the lowercase name is the C Name. The uppercase name is the PL/I, COBOL, and RPG Name.

Table 132. Field Descriptions for an SQLVAR (continued)

C Name [145] COBOL Name PL/I Name RPG Name	Field Data Type	Usage in DESCRIBE and PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, and EXECUTE (set by the user prior to executing the statement)
sqlind SQLIND	pointer	For a select-statement, indicates whether the column was added as a result of using the WITH ROW CHANGE COLUMNS attribute: • -1 ROW CHANGE TOKEN (distinct) • -2 ROW CHANGE TOKEN (not distinct) • -3 RID • -4 RID_BIT Otherwise, reserved.	Contains the address of the indicator variable. Not used if there is no indicator variable (as indicated by an even value of SQLTYPE).
sqlname SQLNAME	VARCHAR(30)	The unqualified name of the column. If the column does not have a name, a string is constructed from the expression and returned. The name is case sensitive and does not contain surrounding delimiters.	Contains the CCSID of the host variable as described in Table 135 on page 1534. For XML data, sqlname can be set as follows to indicate as XML subtype: The length of sqlname is 8 Bytes 1 and 2: Must be X'0000'. Bytes 3 and 4: May contain a CCSID. Bytes 5 and 6 X'0100' XML host variable (XML AS CLOB, XML AS DBCLOB, XML AS BLOB, XML AS CLOB_FILE, XML AS DBCLOB_FILE, XML AS BLOB_FILE) Bytes 7 and 8: Must be X'0000'.

Fields in an occurrence of a secondary SQLVAR

Table 133. Field Descriptions for an Extended SQLVAR

C Name [146] COBOL Name PL/I Name RPG Name	Field Data Type	Usage in DESCRIBE and PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, and EXECUTE (set by the user prior to executing the statement)
len.sqllonglen SQLLONGL SQLLONGLEN	INTEGER	The length attribute of a LOB column. For XML, the value is 0.	The length attribute of a LOB or XML host variable. The database manager ignores the SQLLEN field in the base SQLVAR for these data types. The length attribute indicates the number of bytes for a BLOB, and the number of characters for a CLOB, DBCLOB, or for XML.
*	CHAR(12)	Reserved. Provides boundary alignment for SQLDATALEN.	Reserved. Provides boundary alignment for SQLDATALEN.
*	pointer	Reserved.	Reserved.
sqldatalen SQLDATAL SQLDATALEN	pointer	Not used.	Used only for LOB host variables. If the value of this field is not null, this field points to a four-byte long buffer that contains the actual length of the LOB in bytes (even for DBCLOBs). The SQLDATA field in the matching base SQLVAR then points to the LOB data. If the value of this field is null, the actual length of the LOB is stored in the first four bytes pointed to by the SQLDATA field in the matching base SQLVAR, and the LOB data immediately follows the four-byte length. The actual length indicates the number of bytes for a BLOB or CLOB and the number of double-byte characters for a DBCLOB. Regardless of whether this field is used, field SQLLONGLEN must be set.

146. In this column, the lowercase name is the C Name. The first uppercase name is the PL/I and RPG Name. The second uppercase name is the COBOL Name.

Table 133. Field Descriptions for an Extended SQLVAR (continued)

C Name [146] **COBOL Name** **PL/I Name** **RPG Name**	**Field Data Type**	**Usage in DESCRIBE and PREPARE (set by the database manager)**	**Usage in FETCH, OPEN, CALL, and EXECUTE (set by the user prior to executing the statement)**
sqldatatype_name SQLTNAME SQLDATATYPE-NAME	VARCHAR (30)	The SQLTNAME field of the extended SQLVAR is set to one of the following: • For a distinct type column, the database manager sets this to the fully qualified distinct type name. If the qualified name is longer than 30 bytes, it is truncated. • For a label, the database manager sets this to the first 20 bytes of the label. • For a column name, the database manager sets this to the column name.	Not used.

SQLTYPE and SQLLEN

The following table shows the values that may appear in the SQLTYPE and SQLLEN fields of the SQLDA. In PREPARE and DESCRIBE, an even value of SQLTYPE means the column does not allow nulls, and an odd value means the column does allow nulls.

Note: In an SQLDA used in DESCRIBE or PREPARE statements, an odd value is returned for an expression if one operand is nullable or if the expression may result in a -2 mapping-error null value.

In FETCH, OPEN, CALL, and EXECUTE, an even value of SQLTYPE means no indicator variable is provided, and an odd value means that SQLIND contains the address of an indicator variable.

Table 134. SQLTYPE and SQLLEN values for PREPARE, DESCRIBE, FETCH, OPEN, CALL, or EXECUTE

	For PREPARE and DESCRIBE		For FETCH, OPEN, CALL, and EXECUTE	
SQLTYPE	**COLUMN DATA TYPE**	**SQLLEN**	**HOST VARIABLE DATA TYPE**	**SQLLEN**
384/385	Date [149]	10	Fixed-length character-string representation of a date	Length attribute of the host variable
388/389	Time	8	Fixed-length character-string representation of a time	Length attribute of the host variable
392/393	Timestamp	26	Fixed-length character-string representation of a time stamp	Length attribute of the host variable
396/397	DataLink	Length attribute of the column	DataLink	Length attribute of the host variable
400/401	Not Applicable	Not Applicable	NUL-terminated graphic string	Length attribute of the host variable
404/405	BLOB	0 [148]	BLOB	Not used. [148]
408/409	CLOB	0 [148]	CLOB	Not used. [148]
412/413	DBCLOB	0 [148]	DBCLOB	Not used. [148]
448/449	Varying-length character string	Length attribute of the column	Varying-length character string	Length attribute of the host variable
452/453	Fixed-length character string	Length attribute of the column	Fixed-length character string	Length attribute of the host variable
456/457	Long varying-length character string	Length attribute of the column	Long varying-length character string	Length attribute of the host variable
460/461	Not Applicable	Not Applicable	NUL-terminated character string	Length attribute of the host variable
464/465	Varying-length graphic string	Length attribute of the column	Varying-length graphic string	Length attribute of the host variable
468/469	Fixed-length graphic string	Length attribute of the column	Fixed-length graphic string	Length attribute of the host variable
472/473	Long varying-length graphic string	Length attribute of the column	Long graphic string	Length attribute of the host variable

Table 134. SQLTYPE and SQLLEN values for PREPARE, DESCRIBE, FETCH, OPEN, CALL, or EXECUTE (continued)

	For PREPARE and DESCRIBE		For FETCH, OPEN, CALL, and EXECUTE	
SQLTYPE	**COLUMN DATA TYPE**	**SQLLEN**	**HOST VARIABLE DATA TYPE**	**SQLLEN**
476/477	Not Applicable	Not Applicable	PASCAL L-string	Length attribute of the host variable
480/481	Floating point	4 for single precision, 8 for double precision	Floating point	4 for single precision, 8 for double precision
484/485	Packed decimal	Precision in byte 1; scale in byte 2	Packed decimal	Precision in byte 1; scale in byte 2
488/489	Zoned decimal	Precision in byte 1; scale in byte 2	Zoned decimal	Precision in byte 1; scale in byte 2
492/493	Big integer	8 [147]	Big integer	8
496/497	Large integer	4 [147]	Large integer	4
500/501	Small integer	2 [147]	Small integer	2
504/505	Not Applicable	Not Applicable	DISPLAY SIGN LEADING SEPARATE	Precision in byte 1; scale in byte 2
904/905	ROWID	40	ROWID	40
908/909	Varying-length binary string	Length attribute of the column	Varying-length binary string	Length attribute of the host variable
912/913	Fixed-length binary string	Length attribute of the column	Fixed-length binary string	Length attribute of the host variable
916/917	Not Applicable	Not Applicable	BLOB file reference variable	267
920/921	Not Applicable	Not Applicable	CLOB file reference variable	267
924/925	Not Applicable	Not Applicable	DBCLOB file reference variable	267
960/961	Not Applicable	Not Applicable	BLOB locator	4
964/965	Not Applicable	Not Applicable	CLOB locator	4
968/969	Not Applicable	Not Applicable	DBCLOB locator	4
972	Not Applicable	Not Applicable	Result set locator	8
988/989	XML	0	Not Applicable. Use XML AS CLOB, XML AS DBCLOB, or XML AS BLOB	0
996/997	Not Applicable DECFLOAT(16) DECFLOAT(34)	Not Applicable 8 16	DECFLOAT(7) [150] DECFLOAT(16) DECFLOAT(34)	4 8 16
2452/2453	Not Applicable	Not Applicable	XML locator	4

147. Binary numbers can be represented in the SQLDA with a length of 2, 4, or 8, or with the precision in byte 1 and the scale in byte 2. If the first byte is greater than x'00', it indicates precision and scale.

148. Field SQLLONGLEN in the extended SQLVAR contains the length attribute of the column.

149. Less for *JUL, *YMD, *DMY, and *MDY formats. For more information, see Table 8 on page 84

CCSID values in SQLDATA or SQLNAME

In the OPEN, FETCH, CALL, and EXECUTE statements, the SQLNAME field of the SQLVAR element can be used to specify a CCSID for string host variables. If the SQLNAME field is used to specify a CCSID, the SQLNAME length must be set to 8. In addition, the first 4 bytes of SQLNAME must be set as described in the table below. If no CCSID is specified, the job CCSID is used.

In the DESCRIBE, DESCRIBE TABLE, and PREPARE statements, the SQLDATA field of the SQLVAR element contains the CCSID of the column of the result table if that column is a string column. The CCSID is located in bytes 3 and 4 as described in Table 135.

Table 135. CCSID values for SQLDATA or SQLNAME

Data Type	Encoding Scheme	Bytes 1 & 2	Bytes 3 & 4
Character	SBCS data	X'0000'	ccsid
Character	Mixed data	X'0000'	ccsid
Character	Bit data	X'0000'	65535
Graphic	Not Applicable	X'0000'	ccsid
Any other data type	Not Applicable	Not Applicable	Not Applicable

Unrecognized and unsupported SQLTYPES

The values that appear in the SQLTYPE field of the SQLDA are dependent on the level of data type support available at the sender as well as the receiver of the data. This is particularly important as new data types are added to the product.

New data types may or may not be supported by the sender or receiver of the data and may or may not even be recognized by the sender or receiver of the data. Depending on the situation, the new data type may be returned, or a compatible data type agreed upon by both the sender and receiver of the data may be returned or an error may result.

When the sender and receiver agree to use a compatible data type, the following indicates the mapping that will take place. This mapping will take place when at least one of the sender or receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

Table 136. Compatible Data Types for Unsupported Data Types

Data Type	Compatible Data Type
BIGINT	DECIMAL(19,0)
ROWID	VARCHAR(40) FOR BIT DATA

150. DB2 does not internally store DECFLOAT(7) numbers, but it will support DECFLOAT(7) numbers from applications. A DECFLOAT(7) variable referenced in an SQL statement will be converted to DECFLOAT(16).

INCLUDE SQLDA declarations

This section shows the equivalent INCLUDE SQLDA declaration for C and C++, COBOL, ILE COBOL, PL/I, and ILE RPG.

For C and C++

In C and C++, INCLUDE SQLDA declarations are equivalent to the following:

```
#ifndef SQLDASIZE
struct sqlda
{
       unsigned char  sqldaid[8];
       long           sqldabc;
       short          sqln;
       short          sqld;
       struct sqlvar
       {
              short          sqltype;
              short          sqllen;
              union {
                unsigned char *sqldata;
                long long sqld_result_set_locator;  };
              union {
                short *sqlind;
                long sqld_row_change;
                long sqld_result_set_rows;   };
              struct sqlname
              {
                     short          length;
                     unsigned char  data[30];
              } sqlname;
       } sqlvar[1];
};

struct sqlvar2
       { struct
                {  long           sqllonglen;
                   char           reserve1[28];
                }  len;
                char *sqldatalen;
                struct sqldistinct_type
                       {  short          length;
                          unsigned char  data[30];
                       } sqldatatype_name;
       };

#define  SQLDASIZE(n) (sizeof(struct sqlda)+(n-1) * sizeof(struct sqlvar))
#endif
```

Figure 11. INCLUDE SQLDA Declarations for C and C++

```
                     /*********************************************************************/
                     /* Macros for using the sqlvar2 fields.                              */
                     /*********************************************************************/

                     /*********************************************************************/
                     /*   '2' in the 7th byte of sqldaid indicates a doubled number of    */
                     /*       sqlvar entries.                                             */
                     /*   '3' in the 7th byte of sqldaid indicates a tripled number of    */
                     /*       sqlvar entries.                                             */
                     /*   '4' in the 7th byte of sqldaid indicates a quadrupled number of */
                     /*       sqlvar entries.                                             */
                     /*********************************************************************/
```

```
#define SQLDOUBLED '2'
#define SQLSINGLED ' '

/*********************************************************************/
/* GETSQLDOUBLED(daptr) returns 1 if the SQLDA pointed to by         */
/* daptr has been doubled, or 0 if it has not been doubled.          */
/*********************************************************************/
#define GETSQLDOUBLED(daptr) (((daptr)->sqldaid[6]==  \
 (char) SQLDOUBLED)  ?                                \
    (1)           :      \
    (0)             )

/*********************************************************************/
/* SETSQLDOUBLED(daptr, SQLDOUBLED) sets the 7th byte of sqldaid     */
/* to '2'.                                                           */
/* SETSQLDOUBLED(daptr, SQLSINGLED) sets the 7th byte of sqldaid     */
/* to be a ' '.                                                      */
/*********************************************************************/
#define SETSQLDOUBLED(daptr, newvalue)               \
  (((daptr)->sqldaid[6] =(newvalue)))

/*********************************************************************/
/* GETSQLDALONGLEN(daptr,n) returns the data length of the nth       */
/* entry in the sqlda pointed to by daptr. Use this only if the      */
/* sqlda was doubled or tripled and the nth SQLVAR entry has a       */
/* LOB datatype.                                                     */
/*********************************************************************/
#define GETSQLDALONGLEN(daptr,n) ((long) (((struct sqlvar2 *) \
&((daptr)->sqlvar[(n) +((daptr)->sqld)])) ->len.sqllonglen))

/*********************************************************************/
/* SETSQLDALONGLEN(daptr,n,len) sets the sqllonglen field of the     */
/* sqlda pointed to by daptr to len for the nth entry. Use this only */
/* if the sqlda was doubled or tripled and the nth SQLVAR entry has  */
/* a LOB datatype.                                                   */
/*********************************************************************/
#define SETSQLDALONGLEN(daptr,n,length) {            \
 struct sqlvar2 *var2ptr;                            \
 var2ptr = (struct sqlvar2 *) &((daptr)->sqlvar[(n)+  \
   ((daptr)->sqld)]);                                \
 var2ptr->len.sqllonglen = (long) (length);              \
}

/*********************************************************************/
/* SETSQLDALENPTR(daptr,n,ptr) sets a pointer to the data length for */
/* the nth entry in the sqlda pointed to by daptr.                   */
/* Use this only if the sqlda has been doubled or tripled.           */
/*********************************************************************/
#define SETSQLDALENPTR(daptr,n,ptr) {                \
 struct sqlvar2 *var2ptr;                            \
 var2ptr = (struct sqlvar2 *) &((daptr)->sqlvar[(n)+  \
   ((daptr)->sqld)]);                                \
 var2ptr->sqldatalen = (char *) ptr;                 \
 }
/*********************************************************************/
/* GETSQLDALENPTR(daptr,n) returns a pointer to the data length for  */
/* the nth entry in the sqlda pointed to by daptr. Unlike the inline */
/* value (union sql8bytelen len), which is 8 bytes, the sqldatalen   */
/* pointer field returns a pointer to a long (4 byte) integer.       */
/* If the SQLDATALEN pointer is zero, a NULL pointer is be returned. */
/*                                                                   */
/* NOTE: Use this only if the sqlda has been doubled or tripled.     */
/*********************************************************************/
#define GETSQLDALENPTR(daptr,n) (                    \
  (((struct sqlvar2 *) &(daptr)->sqlvar[(n) +        \
```

```
   (daptr)->sqld])->sqldatalen == NULL) ?           \
  ((long *) NULL ) : ((long *) ((struct sqlvar2 *)   \
 &(daptr)->sqlvar[(n) + (daptr) ->sqld])->sqldatalen))
```

For COBOL and ILE COBOL

In COBOL and ILE COBOL, INCLUDE SQLDA declarations are equivalent to the following:

```
1 SQLDA.
  05 SQLDAID     PIC X(8).
  05 SQLDABC     PIC S9(9) BINARY.
  05 SQLN        PIC S9(4) BINARY.
  05 SQLD        PIC S9(4) BINARY.
  05 SQLVAR OCCURS 0 TO 409 TIMES DEPENDING ON SQLD.
     10 SQLVAR1.
        15 SQLTYPE   PIC S9(4) BINARY.
        15 SQLLEN    PIC S9(4) BINARY.
        15 FILLER  REDEFINES SQLLEN.
           20 SQLPRECISION PIC X.
           20 SQLSCALE     PIC X.
        15 SQLRES    PIC X(12).
        15 SQLDATA   POINTER.
        15 SQL-RESULT-SET-LOCATOR-R REDEFINES SQLDATA.
           20 SQL-RESULT-SET-LOCATOR PIC S9(18) BINARY.
        15 SQLIND   POINTER.
        15 SQL-ROW-CHANGE-SQL-R REDEFINES SQLIND.
           20 SQLD-ROW-CHANGE FIC S9(9) BINARY.
        15 SQL-RESULT-SET-ROWS-R PIC REDEFINES SQLIND.
           20 SQLD-RESULT-SET-ROWS PIC S9(9) BINARY.
        15 SQLNAME.
           49 SQLNAMEL PIC S9(4) BINARY.
           49 SQLNAMEC PIC X(30).
     10 SQLVAR2 REDEFINES SQLVAR1.
        15 SQLVAR2-RESERVED-1 PIC S9(9) BINARY.
        15 SQLLONGLEN REDEFINEDS SQLVAR2-RESERVED-1
                                PIC S9(9) BINARY.
        15 SQLVAR2-RESERVED-2 PIC X(28).
        15 SQLDATALEN POINTER.
        15 SQLDATATYPE-NAME.
           49 SQLDATATYPE_NAMEL PIC S9(4) BINARY.
           49 SQLDATATYPE_NAMEC PIC X(30).
```

Figure 12. INCLUDE SQLDA Declarations for COBOL

For PL/I

In PL/I, INCLUDE SQLDA declarations are equivalent to the following:

```
DCL 1 SQLDA BASED(SQLDAPTR),
      2 SQLDAID     CHAR(8),
      2 SQLDABC     BIN FIXED(31),
      2 SQLN        BIN FIXED,
      2 SQLD        BIN FIXED,
      2 SQLVAR      (99),
        3 SQLTYPE   BIN FIXED,
        3 SQLLEN    BIN FIXED,
        3 SQLRES    CHAR(12),
        3 SQLDATA   PTR,
        3 SQLIND    PTR,
        3 SQLNAME   CHAR(30) VAR,

    1 SQLDA2 BASED(SQLDAPTR),
      2 SQLDAID2    CHAR(8),
      2 SQLDABC2    FIXED(31) BINARY,
      2 SQLN2       FIXED(15) BINARY,
      2 SQLD2       FIXED(15) BINARY,
      2 SQLVAR2     (99),
        3 SQLBIGLEN,
          4 SQLLONGL FIXED(31) BINARY,
          4 SQLRSVDL FIXED(31) BINARY,
        3 SQLDATAL  POINTER,
        3 SQLTNAME  CHAR(30) VAR;

    DECLARE SQLSIZE    FIXED(15) BINARY;
    DECLARE SQLDAPTR   PTR;
    DECLARE SQLDOUBLED CHAR(1)   INITIAL('2') STATIC;
    DECLARE SQLSINGLED CHAR(1)   INITIAL(' ') STATIC;
```

Figure 13. INCLUDE SQLDA Declarations for PL/I

For ILE RPG

In ILE RPG, INCLUDE SQLDA declarations are equivalent to the following:

```
D*      SQL DESCRIPTOR AREA
D SQLDA           DS
D  SQLDAID                       8A
D  SQLDABC                      10I 0
D  SQLN                          5I 0
D  SQLD                          5I 0
D  SQL_VAR                      80A   DIM(SQL_NUM)
D                         33     48*
D                         49     64*
D*
D SQLVAR          DS
D  SQLTYPE                       5I 0
D  SQLLEN                        6I 0
D  SQLRES                       12A
D  SQLINFO1                     16A
D   SQLDATA                       *   OVERLAY(SQLINFO1:1)
D   SQL_RESULT_SET_LOCATOR...
D                               20I 0 OVERLAY(SQLINFO1:1)
D  SQLINFO2                     16A
D   SQLIND                        *   OVERLAY(SQLINFO2:1)
D   SQL_ROW_CHANGE...
D                               10I 0 OVERLAY(SQLINFO2:1)
D   SQL_RESULT_SET_ROWS...
D                               10I 0 OVERLAY(SQLINFO2:1)
D  SQLNAMELEN                    5I 0
D  SQLNAME                      30A
D*  END OF SQLDA
D*  EXTENDED SQLDA
D SQLVAR2         DS
D  SQLLONGL               1      4I 0
D  SQLRSVDL               5     32A
D  SQLDATAL              33     48*
D  SQLTNAMELN            49     50I 0
D  SQLTNAME              51     80A
D*  END OF EXTENDED SQLDA
```

Figure 14. INCLUDE SQLDA Declarations for ILE RPG

The user is responsible for the definition of SQL_NUM. SQL_NUM must be defined as a numeric constant with the dimension required for SQL_VAR.

Since RPG does not support structures within arrays, the SQLDA generates three data structures. The second and third data structures are used to setup/reference the part of the SQLDA which contains the field descriptions.

To set the field descriptions of the SQLDA the program sets up the field description in the subfields of SQLVAR (or SQLVAR2) and then does a MOVEA of SQLVAR (or SQLVAR2) to SQL_VAR, n where n is the number of the field in the SQLDA. This is repeated until all the field descriptions are set.

When the SQLDA field descriptions are to be referenced the user does a MOVEA of SQL_VAR, n to SQLVAR (or SQLVAR2) where n is the number of the field description to be processed.

Appendix E. CCSID values

The tables in this section describe the CCSIDs and conversions provided by the IBM relational database products.

For more information, see “Character conversion” on page 32.

The following list defines the symbols used in the DB2 product column in the following tables:

X Indicates that the conversion tables exist to convert from or to that CCSID. This also implies that this CCSID can be used to tag local data.

C Indicates that conversion tables exist to convert from that CCSID to another CCSID. This also implies that this CCSID cannot be used to tag local data, because the CCSID is in a foreign encoding scheme (for example, a PC-Data CCSID such as 850 cannot be used to tag local data in DB2 for i).

blank Indicates that the specific product does not support the CCSID at all. Such a CCSID must not be used unless interoperability with the specific product is not necessary.

This information is current as of the publishing date of this book for the CCSIDs listed. Additional CCSIDs may have been added since the publishing date and are not in the lists below.

Table 137. Universal Character Set (UTF-8, UTF-16, and UCS-2)

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
1200	UTF-16	X	X	X	X	X	X	X	X	X
1208	UTF-8 Level 3	X	X	X	X	X	X	X	X	X
13488	UCS-2 Level 1	C	X	C *	C *	C *	C *	C *	C *	C *

Note: * In DB2 LUW, 13488 is only used to tag the GRAPHIC column of eucJP and eucTW databases.

Table 138. CCSIDs for EBCDIC Group 1 (Latin-1) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
37	USA, Canada, Netherlands, Portugal, Brazil, Australia, New Zealand	X	X	C	C	C	C	C	C	C
256	Word Processing, Netherlands	X	X							
273	Austria, Germany	X	X	C	C	C	C	C	C	C
274	Belgium	X		C	C	C	C	C	C	C
277	Denmark, Norway	X	X	C	C	C	C	C	C	C
278	Finland, Sweden	X	X	C	C	C	C	C	C	C
280	Italy	X	X	C	C	C	C	C	C	C
284	Spain, Latin America (Spanish)	X	X	C	C	C	C	C	C	C
285	United Kingdom	X	X	C	C	C	C	C	C	C
297	France	X	X	C	C	C	C	C	C	C
500	Belgium, Canada, Switzerland, International Latin-1	X	X	C	C	C	C	C	C	C
871	Iceland	X	X	C	C	C	C	C	C	C
924	Latin-0	X	X							
1047	Latin-0 (with Euro)	X	X							
1140	USA, Canada, Netherlands, Portugal, Brazil, Australia, New Zealand	X	X	C	C	C	C	C	C	C
1141	Austria, Germany	X	X	C	C	C	C	C	C	C
1142	Denmark, Norway	X	X	C	C	C	C	C	C	C
1143	Finland, Sweden	X	X	C	C	C	C	C	C	C
1144	Italy	X	X	C	C	C	C	C	C	C
1145	Spain, Latin America (Spanish)	X	X	C	C	C	C	C	C	C
1146	United Kingdom	X	X	C	C	C	C	C	C	C
1147	France	X	X	C	C	C	C	C	C	C
1148	Belgium, Canada, Switzerland, International Latin-1	X	X	C	C	C	C	C	C	C
1149	Iceland	X	X	C	C	C	C	C	C	C

Table 139. CCSIDs for PC-Data and ISO Group 1 (Latin-1) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
437	USA	X	C	C	C	C	C	C	C	C
819	Latin-1 countries or regions (ISO 8859-1)	X	C	X	X	X	C	X	X	X
850	Latin Alphabet Number 1; Latin-1 countries or regions	X	C	X	C	C	C	C	C	C
858	Latin Alphabet Number 1; Latin-1 countries or regions (with Euro)	X	C							
860	Portugal (850 subset)	X	C	C	C	C	C	C	C	C
861	Iceland	X	C							
863	Canada (850 subset)	X	C	C	C	C	C	C	C	C
865	Denmark, Norway, Finland, Sweden	X	C							
923	Latin-0	X	C	X	X	X	C	C	C	X
1009	IRV 7-bit	X	C							
1010	France 7-bit	X	C							
1011	Germany 7-bit	X	C							
1012	Italy 7-bit	X	C							
1013	United Kingdom 7-bit	X	C							
1014	Spain 7-bit	X	C							
1015	Portugal 7-bit	X	C							
1016	Norway 7-bit	X	C							
1017	Denmark 7-bit	X	C							
1018	Finland and Sweden 7-bit	X	C							
1019	Belgium and Netherlands 7-bit	X	C							
1051	HP Emulation	X	C	C	X	C	C	C	C	C
1252	Windows ** Latin-1	X	C	C	C	C	X	C	C	C
1275	Macintosh ** Latin-1	X	C							
5348	Windows Latin-1 (with Euro)	X	C							

Table 140. CCSIDs for EBCDIC Group 1a (Non-Latin-1 SBCS) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
420	Arabic (Type 4)	X	X	C	C	C	C	C	C	C
423	Greek	X	X	C	C	C	C	C	C	C
424	Hebrew(Type 4)	X	X	C	C	C	C	C	C	C
425	Arabic (Type 5)		X	C	C	C	C	C	C	C
870	Latin-2 Multilingual	X	X	C	C	C	C	C	C	C
875	Greek	X	X	C	C	C	C	C	C	C
880	Cyrillic Multilingual	X	X							
905	Turkey Latin-3 Multilingual	X	X							
918	Urdu	X	X							
1025	Cyrillic Multilingual	X	X	C	C	C	C	C	C	C
1026	Turkey Latin-5	X	X	C	C	C	C	C	C	C
1097	Farsi	X	X							
1112	Baltic Multilingual	X	X	C	C	C	C	C	C	C
1122	Estonia	X	X	C	C	C	C	C	C	C
1123	Ukraine	X	X	C	C	C	C	C	C	C
1137	Devanagari	X	X	C	C	C	C	C	C	C
1153	Latin-2 (with Euro)	X	X	C	C	C	C	C	C	C
1154	Cyrillic (with Euro)	X	X	C	C	C	C	C	C	C
1155	Turkey Latin-5 (with Euro)	X	X	C	C	C	C	C	C	C
1156	Balitic (with Euro)	X	X	C	C	C	C	C	C	C
1157	Estonia (with Euro)	X	X	C	C	C	C	C	C	C
1158	Ukraine (with Euro)	X	X	C	C	C	C	C	C	C
1166	Cyrillic Multilingual (with Euro)		X							
4971	Greek (with Euro)	X	X							
8612	Arabic (Type 5)	X	X							
12708	Arabic (Type 7)		X							
62211	Hebrew (Type 5)		X	C	C	C	C	C	C	C
62224	Arabic (Type 6)		X	C	C	C	C	C	C	C
62229	Hebrew (Type 8)			C	C	C	C	C	C	C
62233	Arabic (Type 8)			C	C	C	C	C	C	C
62234	Arabic (Type 9)			C	C	C	C	C	C	C
62235	Hebrew (Type 6)		X	C	C	C	C	C	C	C
62240	Hebrew (Type 11)			C	C	C	C	C	C	C
62245	Hebrew (Type 10)		X	C	C	C	C	C	C	C
62250	Arabic (Type 12)			C	C	C	C	C	C	C
62251	Arabic (Type 6)		X	C	C	C	C	C	C	C

Table 140. CCSIDs for EBCDIC Group 1a (Non-Latin-1 SBCS) Countries or Regions (continued)

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
String Types:										
4	Visual / Left-to-Right / Shaped / Symmetrical Swapping Off									
5	Implicit / Left-to-Right / Unshaped / Symmetrical Swapping On									
6	Implicit / Right-to-Left / Unshaped / Symmetrical Swapping On									
7	Visual / Contextual / Unshaped / Symmetrical Swapping Off									
8	Visual / Right-to-Left / Shaped / Symmetrical Swapping Off									
9	Visual / Right-to-Left / Shaped / Symmetrical Swapping On									
10	Implicit / Contextual-Left / Unshaped / Symmetrical Swapping On									
11	Implicit / Contextual-Right / Unshaped / Symmetrical Swapping On									
12	Implicit / Right-to-Left / Shaped / Symmetrical Swapping On									

Table 141. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
720	Arabic (MS-Dos)	X	C							
737	Greek (MS-Dos)	X	C	C	C	C	X	C	C	C
775	Baltic (MS-Dos)	X	C							
808	Cyrillic (with Euro)	X								
813	Greek/Latin (ISO 8859-7)	X	C	X	X	C	C	X	C	X
848	Ukraine (with Euro)	X								
849	Belarus (with Euro)	X								
851	Greek	X	C							
852	Latin-2 Multilingual	X	C	C	C	C	C	C	C	C
855	Cyrillic Multilingual	X	C	C	C	C	C	C	C	C
856	Arabic (Type 5)	X	C	X	C	C	C	C	C	C
857	Turkey Latin-5	X	C	C	C	C	C	C	C	C
862	Hebrew (Type 4)	X	C	C	C	C	C	C	C	C
864	Arabic (Type 5)	X	C	C	C	C	C	C	C	C
866	Cyrillic	X	C	C	C	C	C	C	C	C
867	Hebrew (with Euro) (Type 10)	X								
868	Urdu	X	C							
869	Greek	X	C	C	C	C	C	C	C	C
872	Cyrillic Multilingual (with Euro)	X								
878	Russian Internet	X	C							
901	Baltic 8-bit (with Euro)	X	C							
902	Estonia 8-bit (with Euro)	X	C							
912	Latin-2 (ISO 8859-2)	X	C	X	X	C	C	X	C	X
914	Latin-4 (ISO 8859-4)	X	C							
915	Cyrillic Multilingual (ISO 8859-5)	X	C	X	X	C	C	X	C	X
916	Hebrew/Latin (ISO 8859-8) (Type 5)	X	C	X	C	C	C	C	C	X
920	Turkey Latin-5 (ISO 8859-9)	X	C	X	X	C	C	X	C	X
921	Baltic 8-bit (ISO 8859-13)	X	C	X	C	C	C	C	C	C
922	Estonia 8-bit	X	C	X	C	C	C	C	C	C
1008	Arabic 8-bit ISO	X	C							
1046	Arabic (Type 5)	X	C	X	C	C	C	C	C	C
1089	Arabic (ISO 8859-6) (Type 5)	X	C	X	X	C	C	C	C	C
1098	Farsi	X	C							
1124	Ukraine 8-bit ISO	X	C	X	C	C	C	C	C	C
1125	Ukraine	X	C	C	C	C	C	C	C	C
1131	Belarus	X	C	C	C	C	C	C	C	C
1250	Windows Latin-2	X	C	C	C	C	X	C	C	C

Table 141. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries or Regions (continued)

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
1251	Windows Cyrillic	X	C	C	C	C	X	C	C	C
1253	Windows Greek	X	C	C	C	C	X	C	C	C
1254	Windows Turkey	X	C	C	C	C	X	C	C	C
1255	Windows Hebrew (Type 5)	X	C	C	C	C	X	C	C	C
1256	Windows Arabic (Type 5)	X	C	C	C	C	X	C	C	C
1257	Windows Baltic	X	C	C	C	C	X	C	C	C
1280	Macintosh** Greek	X	C							
1281	Macintosh** Turkish	X	C							
1282	Macintosh** Latin-2	X	C							
1283	Macintosh** Cyrillic	X	C							
4909	ISO 8859-7 Greek/Latin (with Euro)	X	C							
4948	Latin-2 Multilingual	X	C							
4951	Cyrillic Multilingual	X	C							
4952	Hebrew	X	C							
4953	Turkey Latin-5	X	C							
4960	Arabic	X	C							
4965	Greek		C							
5346	Windows Latin-2 (with Euro)	X								
5347	Windows Cyrillic (with Euro)	X								
5349	Windows Greek (with Euro)	X								
5350	Windows Turkey (with Euro)	X								
5351	Windows Hebrew (with Euro)	X								
5352	Windows Arabic (with Euro)	X								
5353	Windows Baltic Rim (with Euro)	X								
9056	Arabic (Storage Interchange)	X	C							
62208	Hebrew (Type 4)			X	X	X	X	X	X	X
62209	Hebrew (Type 10)		C	C	C	C	C	C	C	C
62210	Hebrew/Latin (ISO 8859-8) (Type 4)		C	X	X	C	C	C	C	C
62213	Hebrew (Type 5)		C	C	C	C	C	C	C	C
62215	Windows Hebrew (Type 4)		C	C	C	C	X	C	C	C
62218	Arabic (Type 4)		C	C	C	C	C	C	C	C
62220	Hebrew (Type 6)			X	X	X	X	X	C	C
62221	Hebrew (Type 6)		C	C	C	C	C	C	C	C
62222	Hebrew/Latin (ISO 8859-8) (Type 6)		C	X	X	C	C	C	C	C
62223	Windows Hebrew (Type 6)		C	C	C	C	X	C	C	C
62225	Arabic (Type 6)			C	C	C	C	C	C	C

Table 141. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries or Regions (continued)

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
62226	Arabic (Type 6)			X	C	C	C	C	C	C
62227	Arabic (ISO 8859-6) (Type 6)			X	X	C	C	C	C	C
62228	Windows Arabic (Type 6)		C	C	C	C	X	C	C	C
62230	Hebrew (Type 8)			X	X	X	X	X	C	C
62231	Hebrew (Type 8)			C	C	C	C	C	C	C
62232	Hebrew/Latin (ISO 8859-8) (Type 8)			X	X	C	C	C	C	C
62236	Hebrew (Type 10)			X	X	X	X	X	X	X
62238	ISO 8859-8 Hebrew/Latin (Type 10)		C	C	C	C	X	C	C	C
62239	Windows Hebrew (Type 10)		C	C	C	C	X	C	C	C
62241	Hebrew (Type 11)			X	X	X	X	X	X	X
62242	Hebrew (Type 11)			C	C	C	C	C	C	C
62243	Hebrew/Latin (ISO 8859-8) (Type 11)			X	X	C	C	C	C	C
62244	Windows Hebrew (Type 11)			C	C	C	X	C	C	C
62248	Arabic (Type 4)		C							

String Types:

4 Visual / Left-to-Right / Shaped / Symmetrical Swapping Off

5 Implicit / Left-to-Right / Unshaped / Symmetrical Swapping On

6 Implicit / Right-to-Left / Unshaped / Symmetrical Swapping On

7 Visual / Contextual / Unshaped / Symmetrical Swapping Off

8 Visual / Right-to-Left / Shaped / Symmetrical Swapping Off

9 Visual / Right-to-Left / Shaped / Symmetrical Swapping On

10 Implicit / Contextual-Left / Unshaped / Symmetrical Swapping On

11 Implicit / Contextual-Right / Unshaped / Symmetrical Swapping On

12 Implicit / Right-to-Left / Shaped / Symmetrical Swapping On

Table 142. SBCS CCSIDs for EBCDIC Group 2 (DBCS) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
290	Japan Katakana (extended)	X	X	C	C	C	C	C	C	C
833	Korea (extended)	X	X	C	C	C	C	C	C	C
836	Simplified Chinese (extended)	X	X	C	C	C	C	C	C	C
838	Thailand (extended)	X	X	C	C	C	C	C	C	C
1027	Japan English (extended)	X	X	C	C	C	C	C	C	C
1130	Vietnam	X	X	C	C	C	C	C	C	C
1132	Lao	X	X							
1159	Traditional Chinese (extended with Euro)			C	C	C	C	C	C	C
1160	Thai (with Euro)	X	X	C	C	C	C	C	C	C
1164	Vietnam (with Euro)	X	X	C	C	C	C	C	C	C
5123	Japan English (with Euro)	X	X							
8482	Japan Katakana (extended with Euro)	X		C	C	C	C	C	C	C
9030	Thailand (extended)	X	X							
13121	Korea Windows	X	X							
13124	Traditional Chinese	X	X							
28709	Traditional Chinese (extended)	X	X	C	C	C	C	C	C	C

Table 143. SBCS CCSIDs for PC-Data Group 2 (DBCS) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
367	Korea and Simplified Chinese EUC	X	C	X			C			
874	Thailand (extended)	X	C	X	X		X			
891	Korea (non-extended)	C	C							
895	Japan EUC - JISX201 Roman Set	C	C							
896	Japan EUC - JISX201 Katakana Set	C								
897	Japan (non-extended)	C	C							
903	Simplified Chinese (non-extended)	C	C							
904	Traditional Chinese (non-extended)	X	C							
1040	Korea (extended)	C	C							
1041	Japan (extended)	X	C							
1042	Simplified Chinese (extended)	C	C							
1043	Traditional Chinese (extended)	X	C							
1088	Korea (KS Code 5601-89)	X	C							
1114	Traditional Chinese (Big-5)	X	C							
1115	Simplified Chinese GB-Code	X	C							
1126	Korea Windows	X	C							
1129	Vietnam	X	C	X						
1133	Lao ISO	X	C							
1162	Thailand (extended) (180 char) (with Euro)	X								
1163	ISO Vietnam (with Euro)	X								
1258	Vietnam	X	C			X				
4970	Thailand (extended)	X	C							
5210	Traditional Chinese	X	C							
9066	Thailand (extended)	X	C							

Table 144. DBCS CCSIDs for EBCDIC Group 2 (DBCS) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
300	Japan - including 4370 user-defined characters (UDC)	X	X	C	C	C	C	C	C	C
834	Korea - including 1880 UDC	X	X	C	C	C	C	C	C	C
835	Traditional Chinese - including 6204 UDC	X	X	C	C	C	C	C	C	C
837	Simplified Chinese - including 1880 UDC	X	X	C	C	C	C	C	C	C
4396	Japan - including 1880 UDC	X	X	C	C	C	C	C	C	C
4930	Korea Windows	X	X	C	C	C	C	C	C	C
4933	Simplified Chinese	X	X	C	C	C	C	C	C	C
9027	Traditional Chinese (with Euro) - including 6204 UDC	C		C	C	C	C	C	C	C
16684	Japan (with Euro)	X	X	C	C	C	C	C	C	C

Table 145. DBCS CCSIDs for PC-Data Group 2 (DBCS) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
301	Japan - including 1880 UDC	X	C	X	C	C	C	C	C	C
926	Korea - including 1880 UDC	C	C							
927	Traditional Chinese - including 6204 UDC	X	C	C	C	C	C	C	C	C
928	Simplified Chinese - including 1880 UDC	C	C							
941	Japan Windows	X	C	C	C	C	X	C	C	C
947	Traditional Chinese (Big-5)	X	C	X	C	C	X	C	C	C
951	Korea (KS Code 5601-89) - including 1880 UDC	X	C	C	C	C	X	C	C	C
952	Japan (EUC) X208-1990 set	C								
953	Japan (EUC) X212-1990 set	C								
971	Korea (EUC) - including 188 UDC	X	C	X	X	X	C	C	C	C
1351	Japan HP-UX (J15)	X		C	X	C	C	C	C	C
1362	Korea Windows	X	C	C	C	C	X	C	C	C
1380	Simplified Chinese (GB-Code) - including 1880 UDC	X	C	C	C	C	X	X	C	C
1382	Simplified Chinese (EUC) - including 1360 UDC	X	C	X	X	X	C	X	C	C
1385	Traditional Chinese	X	C	C	C	C	X	C	C	C

Table 146. Mixed CCSIDs for EBCDIC Group 2 (DBCS) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
930	Japan Katakana/Kanji (extended) - including 4370 UDC	X	X	C	C	C	C	C	C	C
933	Korea (extended) - including 1880 UDC	X	X	C	C	C	C	C	C	C
935	Simplified Chinese (extended) - including 1880 UDC	X	X	C	C	C	C	C	C	C
937	Traditional Chinese (extended) - including 4370 UDC	X	X	C	C	C	C	C	C	C
939	Japan English/Kanji (extended) - including 4370 UDC	X	X	C	C	C	C	C	C	C
1364	Korea (extended)	X	X	C	C	C	C	C	C	C
1371	Traditional Chinese (extended with Euro) - including 4370 UDC			C	C	C	C	C	C	C
1388	Simplified Chinese	X	X	C	C	C	C	C	C	C
1390	Japan Katakana/Kanji (extended with Euro) - including 4370 UDC	X		C	C	C	C	C	C	C
1399	Japan English (with Euro)	X	X						C	C
5026	Japan Katakana/Kanji (extended) - including 1880 UDC)	X	X	C	C	C	C	C	C	C
5035	Japan English/Kanji (extended) - including 1880 UDC	X	X	C	C	C	C	C	C	C

Table 147. Mixed CCSIDs for PC-Data Group 2 (DBCS) Countries or Regions

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
932	Japan (non-extended) - including 1880 UDC	X	C	X	C	C	C	C	C	C
934	Korea (non-extended) including 1880 UDC		C							
936	Simplified Chinese (non-extended) - including 1880 UDC		C							
938	Traditional Chinese (non-extended) - including 6204 UDC)	X	C	C	C	C	C	C	C	C
942	Japan (extended) - including 1880 UDC	X	C	C	C	C	C	C	C	C
943	Japan NT	X	C	C	C	X	X	C	C	C
944	Korea (extended) - including 1880 UDC		C							
946	Simplified Chinese (extended) - including 1880 UDC		C							
948	Traditional Chinese (extended) - including 6204 UDC	X	C	C	C	C	C	C	C	C
949	Korea (KS Code 5601-89) - including 1880 UDC	X	C	C	C	C	C	C	C	C
950	Traditional Chinese (Big-5)	X	C	X	X	X	X	C	C	X
954	Japan (EUC)		C	X	X	X	C	X	C	X
956	Japan 2022 TCP		C							
957	Japan 2022 TCP		C							
958	Japan 2022 TCP		C							
959	Japan 2022 TCP		C							
964	Traditional Chinese (EUC)		C	X	X	X	C	C	C	C
965	Traditional Chinese 2022 TCP		C							
970	Korea EUC	X	C	X	X	X	C	C	X	X
1363	Korea Windows	X	C	C	C	C	X	C	C	C
1381	Simplified Chinese GB-Code	X	C	C	C	C	X	C	C	C
1383	Simplified Chinese EUC	X	C	X	X	X	C	X	C	X
1386	Simplified Chinese	X	C	X	C	C	X	C	C	C
1392	Simplified Chinese GB18030		C							
5039	Japan HP-UX (J15)	X		C	X	C	C	C	C	C
5050	Japan (EUC)		C							
5052	Japan 2022 TCP		C							
5053	Japan 2022 TCP		C							
5054	Japan 2022 TCP		C							
5055	Japan 2022 TCP		C							

Table 147. Mixed CCSIDs for PC-Data Group 2 (DBCS) Countries or Regions (continued)

CCSID	Description	z/OS	i	AIX	HP	Sun	NT	SCO	SGI	Linux
5307	Japan HP-UX (J15) HISTORICAL	X								
17354	Korea 2022 TCP		C							
25546	Korea 2022 TCP		C							
33722	Japan EUC		C							

Appendix F. DB2 for i catalog views

The views contained in a DB2 for i catalog are described in this section.

The database manager maintains a set of tables containing information about the data in each relational database. These tables are collectively known as the *catalog*. The *catalog tables* contain information about tables, user-defined functions, distinct types, parameters, procedures, packages, views, indexes, aliases, sequences, variables, constraints, triggers, XSR objects, and languages supported by DB2 for i. The catalog also contains information about all relational databases that are accessible from this system.

There are three classes of catalog views:

- IBM i catalog tables and views

  The IBM i catalog tables and views are modeled after the ANS and ISO catalog views, but are not identical to the ANS and ISO catalog views. These tables and views are compatible with prior releases of DB2 for i.

  These tables and views exist in schemas QSYS and QSYS2.

  The catalog tables and views contain information about all tables, parameters, procedures, functions, distinct types, packages, XSR objects, views, indexes, aliases, sequences, variables, triggers, and constraints in the entire relational database. When an SQL schema is created, an additional subset of these views are created into the schema that only contain information about tables, packages, views, indexes, and constraints in that schema.
- ODBC and JDBC catalog views

  The ODBC and JDBC catalog views are designed to satisfy ODBC and JDBC metadata API requests. For example, SQLCOLUMNS. These views are compatible with views on DB2 LUW Version 8. These views will be modified as ODBC or JDBC enhances or modifies their metadata APIs.

  These views exist in schema SYSIBM.
- ANS and ISO catalog views

  The ANS and ISO catalog views are designed to comply with the ANS and ISO SQL standard (the Information Schema catalog views). These views will be modified as the ANS and ISO standard is enhanced or modified.

  There are several columns in these views that are reserved for future standard enhancements.

  There are two versions of these views:

  – The first version of these views exist in schema INFORMATION_SCHEMA[151]. Only rows associated with objects to which the user has some privilege are included in the views. This version is compatible with the ANS and ISO SQL standard.

    If you use of this set of catalog views to prevent users from seeing any information about objects to which they have no privilege, you should revoke privileges to the other catalog views from users and PUBLIC.

  – The second version of these views exist in schema SYSIBM. All rows are included in these views whether or not the user has some privilege to the

151. INFORMATION_SCHEMA is the ANS and ISO SQL standard schema name that contains catalog views. It is a synonym for QSYS2.

objects associated with rows in the views. These views are compatible with views on DB2 LUW Version 8 and will generally perform better than the ANS and ISO views in QSYS2.

For example, assume that a user has the SELECT privilege to the QSYS2.TABLES and SYSIBM.TABLES catalog views but does not have any privilege to a table called WORK.EMPLOYEE. The following SQL statement will not return a result row:

```
SELECT *
  FROM QSYS2.TABLES
  WHERE TABLE_SCHEMA = 'WORK' AND TABLE_NAME = 'EMPLOYEE'
```

However, the following SQL statement will return a result row:

```
SELECT *
  FROM SYSIBM.TABLES
  WHERE TABLE_SCHEMA = 'WORK' AND TABLE_NAME = 'EMPLOYEE'
```

Note: Some of these views use special catalog functions as part of the view definition. These functions exist in SYSIBM and QSYS2. If these functions are used directly in applications, care should be taken because they may be incompatibly changed in future releases or fix packs.

Notes

Names in the Catalog: In general, all names stored in columns of a catalog table are undelimited and case sensitive. For example, assume the following table was created:

```
CREATE TABLE "colname"/"long_table_name"
                        ("long_column_name" CHAR(10),
                         INTCOL INTEGER)
```

If the following select statement is used to return information about the mapping between SQL names and system names, the following select statement could be used:

```
SELECT TABLE_NAME, SYSTEM_TABLE_NAME, COLUMN_NAME, SYSTEM_COLUMN_NAME
    FROM QSYS2/SYSCOLUMNS
    WHERE TABLE_NAME = 'long_table_name' AND
          TABLE_SCHEMA = 'colname'
```

The following rows would be returned:

TABLE_NAME	SYSTEM_TABLE_NAME	COLUMN_NAME	SYSTEM_COLUMN_NAME
long_table_name	"long0001"	long_column_name	LONG_00001
long_table_name	"long0001"	INTCOL	INTCOL

System Names in the Catalog: In general, the longer SQL column names should be used rather than the short system column names. The short system column names for IBM i catalog tables and views are explicitly maintained for compatibility with prior releases and other DB2 products. The short system column names for the ODBC and JDBC catalog views and the ANS and ISO catalog views are not explicitly maintained and may change between releases.

Null Values in the Catalog: If the information in a column is not applicable, the null value is returned. Using the table created above, the following select statement, which queries the NUMERIC_SCALE and the CHARACTER_MAXIMUM_LENGTH, would return the null value when the data was not applicable to the data type of the column.

```
SELECT COLUMN_NAME, NUMERIC_SCALE, CHARACTER_MAXIMUM_LENGTH
  FROM QSYS2/SYSCOLUMNS
  WHERE TABLE_NAME = 'long_table_name'    AND
        TABLE_SCHEMA = 'colname'
```

The following rows would be returned:

COLUMN_NAME	NUMERIC_SCALE	CHARACTER_MAXIMUM_LENGTH
long_column_name	?	10
INTCOL	0	?

Because numeric scale is not valid for a character column, the null value is returned for NUMERIC_SCALE for the "long_column_name" column. Because character length is not valid for a numeric column, the null value is returned for CHARACTER_MAXIMUM_LENGTH for the INTCOL column.

Install and Backup Considerations: Certain catalog tables and any views created over the catalog tables and views should be regularly saved:

- The catalog table QSYS.QADBXRDBD contains relational database information. This table should be regularly saved.
- When an ILE external function or procedure or an SQL function or procedure is restored, information is automatically inserted into these catalog tables. This does not occur for non-ILE external functions and procedures. In order to back up the definitions of non-ILE external functions or procedures, ensure that the catalog tables SYSROUTINES and SYSPARMS are saved or ensure you have a back up of the SQL source statements that were used to create these functions and procedures.
- All catalog views in the QSYS2 or SYSIBM schemas are system objects. This means that any user views created over these catalog views must be deleted when the operating system is installed. All dependent objects must be deleted as well. To avoid this requirement, you can save views before installation and then restore them afterwards.
- Catalog tables in the QSYS library are also system objects. However, the catalog tables in the QSYS library are not deleted during installation. Hence, any views created over these tables are preserved throughout the installation process.

Granting Privileges to Catalog Views: Tables and views in the catalog are like any other database tables and views. If you have authorization, you can use SQL statements to look at data in the catalog views in the same way that you retrieve data from any other table. The tables and views in the catalogs are shipped with the SELECT privilege to PUBLIC. This privilege may be revoked and the SELECT privilege granted to individual users.

QSYS Catalog Tables: Most of the catalog views are based on the following tables in the QSYS library (sometimes called the database cross reference files). These tables are not shipped with the SELECT privilege to PUBLIC and should not be used directly:

QADBCCST	QADBKFLD	QADBXSFLD
QADBFDEP	QADBPKG	QADBXTRIGB
QADBFCST	QADBXRDBD	QADBXTRIGC
QADBIFLD	QADBXREF	QADBXTRIGD

Use of SELECT *: New columns are likely to be added to tables and views in the catalog as new functionality is implemented and as the ISO/ANSI standards evolve. For this reason, it is recommended that SELECT * not be used when accessing catalog tables and views unless your application is prepared to tolerate these new columns.

IBM i catalog tables and views

The IBM i catalog includes the views and tables in the QSYS2 schema displayed in this section.

DB2 for i name	Corresponding ANSI/ISO name	Description
"SYSCATALOGS" on page 1563	CATALOGS	Information about relational databases
"SYSCHKCST" on page 1564	CHECK_CONSTRAINTS	Information about check constraints
"SYSCOLAUTH" on page 1565	COLUMN_PRIVILEGES	Information about column privileges
"SYSCOLUMNS" on page 1566	COLUMNS	Information about column attributes
"SYSCOLUMNS2" on page 1574		Information about column attributes
"SYSCOLUMNSTAT" on page 1583		Information about column statistics
"SYSCST" on page 1586	TABLE_CONSTRAINTS	Information about all constraints
"SYSCSTCOL" on page 1588	CONSTRAINT_COLUMN_USAGE	Information about the columns referenced in a constraint
"SYSCSTDEP" on page 1589	CONSTRAINT_TABLE_USAGE	Information about constraint dependencies on tables
"SYSFIELDS" on page 1590		Information about field procedures
"SYSFUNCS" on page 1594	ROUTINES	Information about user-defined functions
"SYSINDEXES" on page 1598		Information about indexes
"SYSINDEXSTAT" on page 1600		Information about index statistics
"SYSJARCONTENTS" on page 1606		Information about jars for Java routines
"SYSJAROBJECTS" on page 1607		Information about jars for Java routines
"SYSKEYCST" on page 1608	KEY_COLUMN_USAGE	Information about unique, primary, and foreign keys
"SYSKEYS" on page 1609		Information about index keys
"SYSMQTSTAT" on page 1610		Information about materialized query table statistics
"SYSPACKAGE" on page 1614		Information about packages
"SYSPACKAGEAUTH" on page 1616		Information about package privileges
"SYSPACKAGESTAT" on page 1617		Information about package statistics
"SYSPACKAGESTMTSTAT" on page 1623		Information about the SQL statements in packages
"SYSPARMS" on page 1625	PARAMETERS	Information about routine parameters
"SYSPARTITIONDISK" on page 1629		Information about partition disk usage
"SYSPARTITIONINDEXDISK" on page 1631		Information about index disk usage
"SYSPARTITIONINDEXES" on page 1633		Information about partition indexes
"SYSPARTITIONINDEXSTAT" on page 1640		Information about partition index statistics
"SYSPARTITIONMQTS" on page 1645		Information about partition materialized query tables
"SYSPARTITIONSTAT" on page 1649		Information about partition statistics
"SYSPROCS" on page 1652	ROUTINES	Information about procedures
"SYSPROGRAMSTAT" on page 1656		Information about programs, service programs, and modules that contain SQL statements
"SYSPROGRAMSTMTSTAT" on page 1665		Information about SQL statements embedded in programs, service programs, and modules
"SYSREFCST" on page 1668	REFERENTIAL_CONSTRAINTS	Information about referential constraints
"SYSROUTINEAUTH" on page 1669	ROUTINE_PRIVILEGES	Information about routine privileges
"SYSROUTINEDEP" on page 1670	ROUTINE_TABLE_USAGE	Information about function and procedure dependencies
"SYSROUTINES" on page 1671	ROUTINES	Information about functions and procedures

IBM i catalog

DB2 for i name	Corresponding ANSI/ISO name	Description
"SYSSCHEMAAUTH" on page 1677		Information about schema privileges
"SYSSCHEMAS" on page 1678		Information about schemas
"SYSSEQUENCEAUTH" on page 1679	USAGE_PRIVILEGES	Information about sequence privileges
"SYSSEQUENCES" on page 1680		Information about sequences
"SYSTABLEDEP" on page 1683		Information about materialized query table dependencies
"SYSTABLEINDEXSTAT" on page 1684		Information about table index statistics
"SYSTABAUTH" on page 1682	TABLE_PRIVILEGES	Information about table privileges
"SYSTABLES" on page 1689	TABLES	Information about tables and views
"SYSTABLESTAT" on page 1692		Information about table statistics
"SYSTRIGCOL" on page 1695	TRIGGER_COLUMN_USAGE	Information about columns used in a trigger
"SYSTRIGDEP" on page 1696	TRIGGER_TABLE_USAGE	Information about objects used in a trigger
"SYSTRIGGERS" on page 1697	TRIGGERS	Information about triggers
"SYSTRIGUPD" on page 1700	TRIGGERED_UPDATE_COLUMNS	Information about columns in the WHEN clause of a trigger
"SYSTYPES" on page 1701	USER_DEFINED_TYPES	Information about built-in data types and distinct types
"SYSUDTAUTH" on page 1705	UDT_PRIVILEGES	Information about type privileges
"SYSVARIABLEAUTH" on page 1706		Information about global variable privileges
"SYSVARIABLEDEP" on page 1707		Information about objects used in global variables
"SYSVARIABLES" on page 1708		Information about global variables
"SYSVIEWDEP" on page 1714	VIEW_TABLE_USAGE	Information about view dependencies on tables
"SYSVIEWS" on page 1716	VIEWS	Information about definition of a view
"SYSXSROBJECTAUTH" on page 1717	USAGE_PRIVILEGES	Information about XML schema privileges
"XSRANNOTATIONINFO" on page 1718		Information about annotations
"XSROBJECTCOMPONENTS" on page 1719		Information about components in an XML schema
"XSROBJECTHIERARCHIES" on page 1720		Information about XML schema document hierarchy relationships
"XSROBJECTS" on page 1721		Information about XML schemas

SYSCATALOGS

The SYSCATALOGS view contains one row for each relational database that a user can connect to. The following table describes the columns in the SYSCATALOGS view.

Table 148. SYSCATALOGS view

Column Name	System Column Name	Data Type	Description
CATALOG_NAME	LOCATION	VARCHAR(18)	Relational database name.
CATALOG_STATUS	RDBASPSTAT	CHAR(10)	Status of a relational database. **ACTIVE** The relational database is associated with an independent auxiliary storage pool (IASP) that is active, but not yet available. **AVAILABLE** The relational database is available. **VARYOFF** The relational database is associated with an independent auxiliary storage pool (IASP) that is varied off. **VARYON** The relational database is associated with an independent auxiliary storage pool (IASP) that is varied on, but not yet available. **UNKNOWN** The status of the relational database is unknown. The status of remote relational databases is always unknown.
CATALOG_TYPE	RDBTYPE	CHAR(7)	Relational database type. **LOCAL** The relational database is local to this system. **REMOTE** The relational database is on a remote system.
CATALOG_ASPGRP	RDBASPGRP	VARCHAR(10)	Independent auxiliary storage pool (IASP) name.
		Nullable	Contains the null value if the relational database status is UNKNOWN.
CATALOG_ASPNUM	RDBASPNUM	VARCHAR(10)	Independent auxiliary storage pool (IASP) number.
		Nullable	Contains the null value if the relational database status is UNKNOWN.
CATALOG_TEXT	RDBTEXT	VARGRAPHIC(50) CCSID 1200	Relational database text description.
		Nullable	

SYSCHKCST

The SYSCHKCST view contains one row for each check constraint in the SQL schema. The following table describes the columns in the SYSCHKCST view.

Table 149. SYSCHKCST view

Column Name	System Column Name	Data Type	Description
CONSTRAINT_SCHEMA	DBNAME	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint
CHECK_CLAUSE	CHECK	VARGRAPHIC(2000) CCSID 1200 Nullable	Text of the check constraint clause Contains the null value if the check clause cannot be expressed without truncation.
ROUNDING_MODE	DECFLTRND	CHAR(1) Nullable	The rounding mode for the check constraint: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP
SYSTEM_CONSTRAINT_SCHEMA	SYS_CDNAME	CHAR(10)	Name of the system schema containing the constraint.

SYSCOLAUTH

The SYSCOLAUTH view contains one row for every privilege granted on a column. Note that this catalog view cannot be used to determine whether a user is authorized to a column because the privilege to use a column could be acquired through a group user profile or special authority (such as *ALLOBJ). Furthermore, the privilege to use a column is also acquired through privileges granted on the table.

The following table describes the columns in the SYSCOLAUTH view:

Table 150. SYSCOLAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
TABLE_SCHEMA	DBNAME	VARCHAR(128)	Name of the schema
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table
COLUMN_NAME	NAME	VARCHAR(128)	Name of the column
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **UPDATE** The privilege to update the column. **REFERENCES** The privilege to reference the column in a referential constraint.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	Contains the null value.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table or view
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	System name of the column

SYSCOLUMNS

The SYSCOLUMNS view contains one row for every column of each table and view in the SQL schema (including the columns of the SQL catalog).

The following table describes the columns in the SYSCOLUMNS view:

Table 151. SYSCOLUMNS view

Column name	System Column Name	Data Type	Description
COLUMN_NAME	NAME	VARCHAR(128)	Name of the column. This will be the SQL column name if one exists; otherwise, it will be the system column name.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table or view that contains the column. This will be the SQL table or view name if one exists; otherwise, it will be the system table or view name.
TABLE_OWNER	TBCREATOR	VARCHAR(128)	The owner of the table or view.
ORDINAL_POSITION	COLNO	INTEGER	Numeric place of the column in the table or view, ordered from left to right.

Table 151. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
DATA_TYPE	COLTYPE	VARCHAR(8)	Type of column: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **FLOAT** Floating point; FLOAT, REAL, or DOUBLE PRECISION **DECFLOAT** Decimal floating-point **CHAR** Fixed-length character string **VARCHAR** Varying-length character string **CLOB** Character large object string **GRAPHIC** Fixed-length graphic string **VARG** Varying-length graphic string **DBCLOB** Double-byte character large object string **BINARY** Fixed-length binary string **VARBIN** Varying-length binary string **BLOB** Binary large object string **DATE** Date **TIME** Time **TIMESTMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **DISTINCT** Distinct type

Table 151. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
LENGTH	LENGTH	INTEGER	The length attribute of the column; or, in the case of a decimal, numeric, or nonzero precision binary column, its precision: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **Precision of number** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHAR **Maximum length of string** VARCHAR or CLOB **Length of graphic string** GRAPHIC **Maximum length of graphic string** VARGRAPHIC or DBCLOB **Length of string** BINARY **Maximum length of binary string** VARBIN or BLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **Maximum length of datalink URL and comment** DATALINK **40 bytes** ROWID **2147483647 bytes** XML **Same value as the source type** DISTINCT
NUMERIC_SCALE	SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if the column is not decimal, numeric, or binary.
IS_NULLABLE	NULLS	CHAR(1)	If the column can contain null values: **N** No **Y** Yes
IS_UPDATABLE	UPDATES	CHAR(1)	If the column can be updated: **N** No **Y** Yes

Table 151. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement.
		Nullable	Contains the null value if there is no long comment.
HAS_DEFAULT	DEFAULT	CHAR(1)	If the column has a default value (DEFAULT clause or null capable): **N** No **Y** Yes **A** The column has a ROWID data type and the GENERATED ALWAYS attribute. **D** The column has a ROWID data type and the GENERATED BY DEFAULT attribute. **E** The column is defined with the FOR EACH ROW ON UPDATE and the GENERATED ALWAYS attribute. **F** The column is defined with the FOR EACH ROW ON UPDATE and the GENERATED BY DEFAULT attribute. **I** The column is defined with the AS IDENTITY and GENERATED ALWAYS attributes. **J** The column is defined with the AS IDENTITY and GENERATED BY DEFAULT attributes. If the column is for a view, N is returned.
COLUMN_HEADING	LABEL	VARGRAPHIC(60) CCSID 1200	A character string supplied with the LABEL statement (column headings)
		Nullable	Contains the null value if there is no column heading.

Table 151. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
STORAGE	STORAGE	INTEGER	The storage requirements for the column: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **(Precision/2) + 1** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHAR or BINARY **Maximum length of string + 2** VARCHAR or VARBIN **Maximum length of string + 29** CLOB or BLOB **Length of string * 2** GRAPHIC **Maximum length of string * 2 + 2** VARGRAPHIC **Maximum length of string * 2 + 29** DBCLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **Maximum length of datalink URL and comment + 24** DATALINK **42 bytes** ROWID **2147483647 bytes + 29** XML **Same value as the source type** DISTINCT **Note:** This column supplies the storage requirements for all data types.
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	The precision of all numeric columns. **Note:** This column supplies the precision of all numeric data types, including decimal floating-point and single-and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the column is not numeric.

Table 151. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
CCSID	CCSID	INTEGER Nullable	The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB, XML, and DATALINK columns. Contains 65535 if the column is a BINARY, VARBIN, BLOB, or ROWID. Contains the null value if the column is a numeric data type.
TABLE_SCHEMA	DBNAME	VARCHAR(128)	The name of the SQL schema containing the table or view.
COLUMN_DEFAULT	DFTVALUE	VARGRAPHIC(2000) CCSID 1200 Nullable	The default value of a column, if one exists. If the default value of the column cannot be represented without truncation, then the value of the column is the string 'TRUNCATED'. The default value is stored in character form. The following special values also exist: **CURRENT_DATE** The default value is the current date. **CURRENT_TIME** The default value is the current time. **CURRENT_TIMESTAMP** The default value is the current timestamp. **NULL** The default value is the null value and DEFAULT NULL was explicitly specified. **USER** The default value is the current job user. Contains the null value if: • The column has no default value. For example, if the column has an IDENTITY attribute, is a row ID, or is a row change timestamp column; or • A DEFAULT value was not explicitly specified.
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER Nullable	Maximum length of the string for binary, character, and graphic string and XML data types. Contains the null value if the column is not a string.
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER Nullable	Number of bytes for binary, character, and graphic string and XML data types. Contains the null value if the column is not a string.
NUMERIC_PRECISION_RADIX	RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the column is not numeric.

Table 151. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
DATETIME_PRECISION	DATPRC	INTEGER	The fractional part of a date, time, or timestamp.
		Nullable	**0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if the column is not a date, time, or timestamp.
COLUMN_TEXT	LABELTEXT	VARGRAPHIC(50) CCSID 1200	A character string supplied with the LABEL statement (column text)
		Nullable	Contains the null value if the column has no column text.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	The system name of the column
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	The system name of the table or view
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	The system name of the schema
USER_DEFINED_TYPE_SCHEMA	TYPESCHEMA	VARCHAR(128)	The name of the schema if this is a distinct type.
		Nullable	Contains the null value if the column is not a distinct type.
USER_DEFINED_TYPE_NAME	TYPENAME	VARCHAR(128)	The name of the distinct type.
		Nullable	Contains the null value if the column is not a distinct type.
IS_IDENTITY	IDENTITY	VARCHAR(3)	This column identifies whether the column is an identity column. **NO** The column is not an identity column. **YES** The column is an identity column.
IDENTITY_GENERATION	GENERATED	VARCHAR(10)	This column identifies whether the column is GENERATED ALWAYS or GENERATED BY DEFAULT.
		Nullable	**ALWAYS** The column value is always generated. **BY DEFAULT** The column value is generated by default. Contains the null value if the column is not a ROWID, IDENTITY, or row change timestamp column.
IDENTITY_START	START	DECIMAL(31,0)	Starting value of the identity column.
		Nullable	Contains the null value if the column is not an IDENTITY column.
IDENTITY_INCREMENT	INCREMENT	DECIMAL(31,0)	Increment value of the identity column.
		Nullable	Contains the null value if the column is not an IDENTITY column.
IDENTITY_MINIMUM	MINVALUE	DECIMAL(31,0)	Minimum value of the identity column.
		Nullable	Contains the null value if the column is not an IDENTITY column.
IDENTITY_MAXIMUM	MAXVALUE	DECIMAL(31,0)	Maximum value of the identity column.
		Nullable	Contains the null value if the column is not an IDENTITY column.

Table 151. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
IDENTITY_CYCLE	CYCLE	VARCHAR(3) Nullable	This column identifies whether the identity column values will continue to be generated after the minimum or maximum value has been reached. **NO** Values will not continue to be generated. **YES** Values will continue to be generated. Contains the null value if the column is not an IDENTITY column.
IDENTITY_CACHE	CACHE	INTEGER Nullable	Specifies the number of identity values that may be preallocated for faster access. Zero indicates that the values will not be preallocated. Contains the null value if the column is not an IDENTITY column.
IDENTITY_ORDER	ORDER	VARCHAR(3) Nullable	Specifies whether the identity values must be generated in order of the request. **NO** Values do not need to be generated in order of the request. **YES** Values must be generated in order of the request. Contains the null value if the column is not an IDENTITY column.
COLUMN_EXPRESSION	EXPRESSION	DBCLOB(2097152) CCSID 1200 Nullable	If the column is an expression, contains the expression. Contains the null value if the column is not an expression.
HIDDEN	HIDDEN	CHAR(1)	Specifies whether the column is included in an implicit column list. **P** Partially hidden. **N** Not hidden.
HAS_FIELDPROC	FLDPROC	CHAR(1)	Specifies whether the column has a field procedure. **N** Column does not have a field procedure. **Y** Column has a field procedure.

SYSCOLUMNS2

The SYSCOLUMNS2 view contains one row for every column of each table and view in the SQL schema (including the columns of the SQL catalog).

For information related to a single table or view, a query that uses SYSCOLUMNS2 will typically perform better than querying SYSCOLUMNS. SYSCOLUMNS2 also contains a few more column attributes than SYSCOLUMNS.

The following table describes the columns in the SYSCOLUMNS2 view:

Table 152. SYSCOLUMNS2 view

Column name	System Column Name	Data Type	Description
COLUMN_NAME	NAME	VARCHAR(128)	Name of the column. This will be the SQL column name if one exists; otherwise, it will be the system column name.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table or view that contains the column. This will be the SQL table or view name if one exists; otherwise, it will be the system table or view name.
TABLE_OWNER	TBCREATOR	VARCHAR(128)	The owner of the table or view.
ORDINAL_POSITION	COLNO	INTEGER	Numeric place of the column in the table or view, ordered from left to right.

Table 152. SYSCOLUMNS2 view (continued)

Column name	System Column Name	Data Type	Description
DATA_TYPE	COLTYPE	VARCHAR(8)	Type of column: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **FLOAT** Floating point; FLOAT, REAL, or DOUBLE PRECISION **DECFLOAT** Decimal floating-point **CHAR** Fixed-length character string **VARCHAR** Varying-length character string **CLOB** Character large object string **GRAPHIC** Fixed-length graphic string **VARG** Varying-length graphic string **DBCLOB** Double-byte character large object string **BINARY** Fixed-length binary string **VARBIN** Varying-length binary string **BLOB** Binary large object string **DATE** Date **TIME** Time **TIMESTMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **DISTINCT** Distinct type

Table 152. SYSCOLUMNS2 view (continued)

Column name	System Column Name	Data Type	Description
LENGTH	LENGTH	INTEGER	The length attribute of the column; or, in the case of a decimal, numeric, or nonzero precision binary column, its precision: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **Precision of number** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHAR **Maximum length of string** VARCHAR or CLOB **Length of graphic string** GRAPHIC **Maximum length of graphic string** VARGRAPHIC or DBCLOB **Length of string** BINARY **Maximum length of binary string** VARBIN or BLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **Maximum length of datalink URL and comment** DATALINK **40 bytes** ROWID **2147483647 bytes** XML **Same value as the source type** DISTINCT
NUMERIC_SCALE	SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if the column is not decimal, numeric, or binary.
IS_NULLABLE	NULLS	CHAR(1)	If the column can contain null values: **N** No **Y** Yes
IS_UPDATABLE	UPDATES	CHAR(1)	If the column can be updated: **N** No **Y** Yes

Table 152. SYSCOLUMNS2 view (continued)

Column name	System Column Name	Data Type	Description
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement.
		Nullable	Contains the null value if there is no long comment.
HAS_DEFAULT	DEFAULT	CHAR(1)	If the column has a default value (DEFAULT clause or null capable): **N** No **Y** Yes **A** The column has a ROWID data type and the GENERATED ALWAYS attribute. **D** The column has a ROWID data type and the GENERATED BY DEFAULT attribute. **E** The column is defined with the FOR EACH ROW ON UPDATE and the GENERATED ALWAYS attribute. **F** The column is defined with the FOR EACH ROW ON UPDATE and the GENERATED BY DEFAULT attribute. **I** The column is defined with the AS IDENTITY and GENERATED ALWAYS attributes. **J** The column is defined with the AS IDENTITY and GENERATED BY DEFAULT attributes. If the column is for a view, N is returned.
COLUMN_HEADING	LABEL	VARGRAPHIC(60) CCSID 1200	A character string supplied with the LABEL statement (column headings)
		Nullable	Contains the null value if there is no column heading.

Table 152. SYSCOLUMNS2 view (continued)

Column name	System Column Name	Data Type	Description
STORAGE	STORAGE	INTEGER	The storage requirements for the column: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **(Precision/2) + 1** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHAR or BINARY **Maximum length of string + 2** VARCHAR or VARBIN **Maximum length of string + 29** CLOB or BLOB **Length of string * 2** GRAPHIC **Maximum length of string * 2 + 2** VARGRAPHIC **Maximum length of string * 2 + 29** DBCLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **Maximum length of datalink URL and comment + 24** DATALINK **42 bytes** ROWID **2147483647 bytes + 29** XML **Same value as the source type** DISTINCT **Note:** This column supplies the storage requirements for all data types.
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	The precision of all numeric columns. **Note:** This column supplies the precision of all numeric data types, including decimal floating-point and single-and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the column is not numeric.

Table 152. SYSCOLUMNS2 view (continued)

Column name	System Column Name	Data Type	Description
CCSID	CCSID	INTEGER Nullable	The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB, XML, and DATALINK columns. Contains 65535 if the column is a BINARY, VARBIN, BLOB, or ROWID. Contains the null value if the column is a numeric data type.
TABLE_SCHEMA	DBNAME	VARCHAR(128)	The name of the SQL schema containing the table or view.
COLUMN_DEFAULT	DFTVALUE	VARGRAPHIC(2000) CCSID 1200 Nullable	The default value of a column, if one exists. If the default value of the column cannot be represented without truncation, then the value of the column is the string 'TRUNCATED'. The default value is stored in character form. The following special values also exist: **CURRENT_DATE** The default value is the current date. **CURRENT_TIME** The default value is the current time. **CURRENT_TIMESTAMP** The default value is the current timestamp. **NULL** The default value is the null value and DEFAULT NULL was explicitly specified. **USER** The default value is the current job user. Contains the null value if: • The column has no default value. For example, if the column has an IDENTITY attribute, is a row ID, or is a row change timestamp column; or • A DEFAULT value was not explicitly specified.
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER Nullable	Maximum length of the string for binary, character, and graphic string and XML data types. Contains the null value if the column is not a string.
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER Nullable	Number of bytes for binary, character, and graphic string and XML data types. Contains the null value if the column is not a string.
NUMERIC_PRECISION_RADIX	RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the column is not numeric.

Table 152. SYSCOLUMNS2 view (continued)

Column name	System Column Name	Data Type	Description
DATETIME_PRECISION	DATPRC	INTEGER	The fractional part of a date, time, or timestamp.
		Nullable	**0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if the column is not a date, time, or timestamp.
COLUMN_TEXT	LABELTEXT	VARGRAPHIC(50) CCSID 1200	A character string supplied with the LABEL statement (column text)
		Nullable	Contains the null value if the column has no column text.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	The system name of the column
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	The system name of the table or view
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	The system name of the schema
USER_DEFINED_TYPE_SCHEMA	TYPESCHEMA	VARCHAR(128)	The name of the schema if this is a distinct type.
		Nullable	Contains the null value if the column is not a distinct type.
USER_DEFINED_TYPE_NAME	TYPENAME	VARCHAR(128)	The name of the distinct type.
		Nullable	Contains the null value if the column is not a distinct type.
IS_IDENTITY	IDENTITY	VARCHAR(3)	This column identifies whether the column is an identity column. **NO** The column is not an identity column. **YES** The column is an identity column.
IDENTITY_GENERATION	GENERATED	VARCHAR(10)	This column identifies whether the column is GENERATED ALWAYS or GENERATED BY DEFAULT.
		Nullable	**ALWAYS** The column value is always generated. **BY DEFAULT** The column value is generated by default. Contains the null value if the column is not a ROWID, IDENTITY, or row change timestamp column.
IDENTITY_START	START	DECIMAL(31,0)	Starting value of the identity column.
		Nullable	Contains the null value if the column is not an IDENTITY column.
IDENTITY_INCREMENT	INCREMENT	DECIMAL(31,0)	Increment value of the identity column.
		Nullable	Contains the null value if the column is not an IDENTITY column.
IDENTITY_MINIMUM	MINVALUE	DECIMAL(31,0)	Minimum value of the identity column.
		Nullable	Contains the null value if the column is not an IDENTITY column.
IDENTITY_MAXIMUM	MAXVALUE	DECIMAL(31,0)	Maximum value of the identity column.
		Nullable	Contains the null value if the column is not an IDENTITY column.

Table 152. SYSCOLUMNS2 view (continued)

Column name	System Column Name	Data Type	Description
IDENTITY_CYCLE	CYCLE	VARCHAR(3) Nullable	This column identifies whether the identity column values will continue to be generated after the minimum or maximum value has been reached. **NO** Values will not continue to be generated. **YES** Values will continue to be generated. Contains the null value if the column is not an IDENTITY column.
IDENTITY_CACHE	CACHE	INTEGER Nullable	Specifies the number of identity values that may be preallocated for faster access. Zero indicates that the values will not be preallocated. Contains the null value if the column is not an IDENTITY column.
IDENTITY_ORDER	ORDER	VARCHAR(3) Nullable	Specifies whether the identity values must be generated in order of the request. **NO** Values do not need to be generated in order of the request. **YES** Values must be generated in order of the request. Contains the null value if the column is not an IDENTITY column.
COLUMN_EXPRESSION	EXPRESSION	DBCLOB(2097152) CCSID 1200 Nullable	If the column is an expression, contains the expression. Contains the null value if the column is not an expression.
HIDDEN	HIDDEN	CHAR(1)	Specifies whether the column is included in an implicit column list. **P** Partially hidden. **N** Not hidden.
HAS_FIELDPROC	FLDPROC	CHAR(1)	Specifies whether the column has a field procedure. **N** Column does not have a field procedure. **Y** Column has a field procedure.
INLINE_LENGTH	ALLOCATE	INTEGER Nullable	Specifies the allocated length (ALLOCATE) for a varying length column. Contains the null value if the column is not varying length.
NORMALIZE	NORMALIZE	CHAR(1) Nullable	Specifies whether the column data should be normalized when passed from the application. **0** Column should not be normalized. **1** Column should be normalized. Contains the null value if the column does not contain Unicode data.
DATALINK_LINK_CONTROL	DL_LINKC	CHAR(1) Nullable	Specifies whether a check will be performed to determine if the DATALINK column's linked files exist. **0** No check will be performed. **1** A check will be performed. Contains the null value if the data type of the column is not DATALINK.

Table 152. SYSCOLUMNS2 view (continued)

Column name	System Column Name	Data Type	Description
DATALINK_INTEGRITY	DL_INTEG	CHAR(1) Nullable	Specifies the level of integrity of the link between the DATALINK value and the linked files. **0** ALL Contains the null value if the data type of the column is not DATALINK or if the datalink has NO LINK CONTROL.
DATALINK_READ_PERMISSION	DL_READP	CHAR(1) Nullable	Specifies how permission to read the file specified in the DATALINK value is determined. **0** FS **1** DB Contains the null value if the data type of the column is not DATALINK or if the datalink has NO LINK CONTROL.
DATALINK_WRITE_PERMISSION	DL_WRITEP	CHAR(1) Nullable	Specifies how permission to write to the file specified in the DATALINK value is determined. **0** FS **1** BLOCKED Contains the null value if the data type of the column is not DATALINK or if the datalink has NO LINK CONTROL.
DATALINK_RECOVERY	DL_RECOVER	CHAR(1) Nullable	Specifies whether point in time recovery of the linked files of the DATALINK column is supported. **0** NO Contains the null value if the data type of the column is not DATALINK or if the datalink has NO LINK CONTROL.
DATALINK_UNLINK_CONTROL	DL_UNLINKC	CHAR(1) Nullable	Specifies the action the DataLink File Manager will take when a file is unlinked. **0** RESTORE **1** DELETE Contains the null value if the data type of the column is not DATALINK or if the datalink has NO LINK CONTROL.
DDS_TYPE	DDS_TYPE	CHAR(1) Nullable	Specifies the Data Description Specification (DDS) data type for the column.
SECURE	SECURE	CHAR(1) Nullable	Specifies whether the column contains data the should be secured in a database monitor or plan cache. **0** The column does not contain data that needs to be secured in a database monitor or plan cache. **1** The column contains data that needs to be secured in a database monitor or plan cache.

SYSCOLUMNSTAT

The SYSCOLUMNSTAT view contains one row for every column in a table partition or table member that has a column statistics collection. If the table is a distributed table, the partitions that reside on other database nodes are not contained in this catalog view.

They are contained in the catalog views of the other database nodes. The following table describes the columns in the SYSCOLUMNSTAT view:

Table 153. SYSCOLUMNSTAT view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
TABLE_PARTITION	TABPART	VARCHAR(128)	Name of the table partition or member.
PARTITION_TYPE	PARTTYPE	CHAR(1)	The type of the table partitioning: **blank** The table is not partitioned. **H** This is data hash partitioning. **R** This is data range partitioning. **D** This is distributed database hash partitioning.
PARTITION_NUMBER	PARTNBR	INTEGER Nullable	The partition number of this partition. If the table is a distributed table, contains null.
NUMBER_DISTRIBUTED_PARTITIONS	DSTPARTS	INTEGER Nullable	If the table is a distributed table, contains the total number of partitions. If the table is not a distributed table, contains null.
NUMBER_COLUMN_NAMES	NBRCOLS	INTEGER	Number of column names in this collection. If only individual column statistics are wanted, only select rows where NUMBER_COLUMN_NAMES is one. Currently, only one name is returned.
COLUMN_NAME	COLNAME	VARCHAR(1280)	Name of the column(s). Up to 10 columns may be returned. Currently, only one name is returned.
NUMBER_DISTINCT_VALUES	COLCARD	BIGINT	Number of distinct values in the column. Contains -1 if statistics are not collected.
HIGH2KEY	HIGH2KEY	VARCHAR(254)	Not applicable for DB2 for i. Contains the empty string.
LOW2KEY	LOW2KEY	VARCHAR(254)	Not applicable for DB2 for i. Contains the empty string.
AVERAGE_COLUMN_LENGTH	AVGCOLLEN	INTEGER	Not applicable for DB2 for i. Will always be -1.
NUMBER_NULLS	NUMNULLS	BIGINT	The estimated number of NULL values. -1 if statistics are not collected.
SUB_COUNT	SUB_COUNT	SMALLINT	Not applicable for DB2 for i. Will always be -1.
SUB_DELIM_LENGTH	SUBDLENGTH	SMALLINT	Not applicable for DB2 for i. Will always be -1.
NUMBER_HISTOGRAM_RANGES	NQUANTILES	INTEGER	Number of histogram ranges available for this statistics collection. The actual histogram range values can be obtained using the List Statistics Collection Details (QDBSTLDS, QdbstListDetailStatistics) API. Contains -1 if statistics are not collected.
NUMBER_MOST_FREQUENT_VALUES	NMOSTFREQ	INTEGER	Number of most frequent values available. The actual most frequent values can be obtained using the List Statistics Collection Details (QDBSTLDS, QdbstListDetailStatistics) API. Contains -1 if statistics are not collected.

Table 153. SYSCOLUMNSTAT view (continued)

Column name	System Column Name	Data Type	Description
AVGDISTINCTPERPAGE	AVGDSTPAGE	DOUBLE Nullable	Not applicable for DB2 for i. Will always be NULL.
PAGEVARIANCERATIO	PVARRATIO	DOUBLE Nullable	Not applicable for DB2 for i. Will always be NULL.
STATISTICS_NAME	STATNAME	VARCHAR(128) Nullable	Unique name of this statistics collection for this table partition. NULL if statistics are not collected.
INTERNAL_STATISTICS_ID	STATID	VARCHAR(16) FOR BIT DATA Nullable	Internal statistics identifier of this statistics collection for this table partition. NULL if statistics are not collected.
STATISTIC_CREATED	STATCREATE	TIMESTAMP Nullable	Timestamp when the statistics collection was created. NULL if statistics are not collected.
STATISTIC_CREATOR	STATCUSER	VARCHAR(128) Nullable	User that created the statistic collection. *SYS if the system created the statistic collection. NULL if statistics are not collected.
STATISTIC_LAST_UPDATED	UPDATEDTS	TIMESTAMP Nullable	Timestamp when the statistics collection was last updated. NULL if statistics are not collected.
STATISTIC_UPDATER	STATUUSER	VARCHAR(128) Nullable	User that last updated the statistic collection. *SYS if the system automatically updated the statistic collection. NULL if statistics are not collected.
STATISTICS_SIZE	STATSIZE	BIGINT Nullable	Size of the statistics collection for this table partition. NULL if statistics are not collected.
AGING_MODE	AGING_MODE	VARCHAR(10)	Indicates whether the system can automatically age or remove statistics collections for this table partition. ***SYS** The statistic collection will be automatically refreshed or removed by the system when necessary. ***USER** The statistic collection will only be refreshed or removed when explicitly requested by the user.
AGING_STATUS	AGING_STS	CHAR(1) Nullable	Indicates how current the statistics collection is for this table partition. **0** There are no indications that the statistics data needs to be refreshed. **1** There are indications that the statistics data needs to be refreshed. NULL if statistics are not collected.
BLOCK_OPTION	BLKOPTION	CHAR(1)	Indicates whether automatic statistics collection create requests are allowed for this table partition. **0** Automatic system initiated statistics collections are not blocked. **1** Automatic system initiated statistics collections are blocked.
CURRENT_LAST_CHANGE	UPDATED	TIMESTAMP Nullable	Timestamp when the data in the table partition was last changed. NULL if statistics are not collected.
CURRENT_ROWS	CURROWS	BIGINT Nullable	Current number of valid rows in the table partition. NULL if statistics are not collected.

Table 153. SYSCOLUMNSTAT view (continued)

Column name	System Column Name	Data Type	Description
CURRENT_DELETED_ROWS	CURDELROWS	BIGINT Nullable	Current number of deleted rows in the table partition. NULL if statistics are not collected.
CURRENT_DATA_CHANGES	CURDATACHG	BIGINT Nullable	The number of inserts, updates, and deletes that have ever occurred to this table partition. NULL if statistics are not collected.
STATISTICS_ROWS	STATROWS	BIGINT Nullable	Number of valid rows in the table partition at the time the statistic was collected. NULL if statistics are not collected.
STATISTICS_DELETED_ROWS	STATDELROW	BIGINT Nullable	Number of deleted rows in the table partition at the time the statistic was collected. NULL if statistics are not collected.
STATISTICS_DATA_CHANGES	STATDATCHG	BIGINT Nullable	Number of inserts, updates, and deletes that had occurred to the table partition at the time the statistic was collected. NULL if statistics are not collected.
TRANSLATION_ATTRIBUTES	TRANSATRS	VARCHAR(10) Nullable	Indicates the type of translations that were used on data values when the statistic was collected. **0** Unique weight translation. **1** Shared weight translation. **9** No translation. If multiple columns are used in this collection, multiple translations are possible. Currently, only one translation is returned.
TRANSLATION_TABLES	TRANSTBLS	VARCHAR(210) Nullable	Qualified names of the translation tables, if translation tables were used on the statistic collection. The empty string is returned if no translation table was used. NULL if statistics are not collected. If multiple columns are used in this collection, multiple translation tables are possible. Currently, only one translation table is returned.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_MEMBER	SYS_MNAME	CHAR(10)	System member name.
SYSTEM_COLUMN_NAME	SYS_CNAME	VARCHAR(100)	System column name. An array of up to 10 names are possible. Currently, only one name is returned.

SYSCST

The SYSCST view contains one row for each constraint in the SQL schema.

The following table describes the columns in the SYSCST view:

Table 154. SYSCST view

Column Name	System Column Name	Data Type	Description
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
CONSTRAINT_TYPE	TYPE	VARCHAR(11)	Constraint Type • CHECK • UNIQUE • PRIMARY KEY • FOREIGN KEY
TABLE_SCHEMA	TDBNAME	VARCHAR(128)	Name of the schema containing the table.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table which the constraint is created over. This will be the SQL table name if it exists; otherwise, it will be the system table name.
IS_DEFERRABLE	ISDEFER	VARCHAR(3)	Indicates whether the constraint checking can be deferred. Will always be 'NO'.
INITIALLY_DEFERRED	INITDEFER	VARCHAR(3)	Indicates whether the constraint was defined as initially deferred. Will always be 'NO'.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the table.
CONSTRAINT_KEYS	COLCOUNT	SMALLINT Nullable	Specifies the number of key columns if this is a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint. Contains the null value if the constraint is a CHECK constraint.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
CONSTRAINT_STATE	CST_STATE	VARCHAR(11)	Indicates whether the constraint is established or defined: **ESTABLISHED** The referential constraint is established. The parent table exists. **DEFINED** The referential constraint is defined. The parent table does not exist.
ENABLED	ENABLED	VARCHAR(3) Nullable	Indicates whether the constraint is enabled: **NO** The constraint is disabled. **YES** The constraint is enabled. Contains the null value if the constraint is defined or is a unique constraint.
CHECK_PENDING	CHECKFLAG	VARCHAR(3) Nullable	Indicates whether the constraint is in check pending state: **NO** The constraint is not in check pending. **YES** The constraint is in check pending. Contains the null value if the constraint is defined, disabled, or is a unique constraint.
CONSTRAINT_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	A character string provided with the LABEL statement. Contains the null value if there is no label.

Table 154. SYSCST view (continued)

Column Name	System Column Name	Data Type	Description
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement.
		Nullable	Contains the null value if there is no long comment.
SYSTEM_CONSTRAINT_SCHEMA	SYS_CDNAME	CHAR(10)	Name of the system schema containing the constraint.

SYSCSTCOL

The SYSCSTCOL view records the columns on which constraints are defined. There is one row for every column in a unique, primary key, and check constraint and the referencing columns of a referential constraint.

The following table describes the columns in the SYSCSTCOL view:

Table 155. SYSCSTCOL view

Column Name	System Column Name	Data Type	Description
TABLE_SCHEMA	TDBNAME	VARCHAR(128)	Name of the SQL schema that contains the table the constraint is dependent on.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table the constraint is dependent on. This is the SQL table name if it exists; otherwise, it is the system table name.
COLUMN_NAME	COLUMN	VARCHAR(128)	Column that the constraint was created over. This is the SQL column name if it exists; otherwise, it is the system column name.
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema of the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	System name of the column.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the table.
SYSTEM_CONSTRAINT_SCHEMA	SYS_CDNAME	CHAR(10)	Name of the system schema containing the constraint.

SYSCSTDEP

The SYSCSTDEP view records the tables on which constraints are defined.

The following table describes the columns in the SYSCSTDEP view:

Table 156. SYSCSTDEP view

Column Name	System Column Name	Data Type	Description
TABLE_SCHEMA	TDBNAME	VARCHAR(128)	Name of the SQL schema that contains the table on which the constraint is dependent
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table on which the constraint is dependent. This is the SQL table name if it exists otherwise it is the system table name.
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema of the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the table.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
SYSTEM_CONSTRAINT_SCHEMA	SYS_CDNAME	CHAR(10)	Name of the system schema containing the constraint.

SYSFIELDS

The SYSFIELDS view contains one row for every column that has a field procedure.

The column attributes in SYSFIELDS describe the internal column attributes defined by the field procedure. The following table describes the columns in the SYSFIELDS view:

Table 157. SYSFIELDS view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	DBNAME	VARCHAR(128)	The name of the SQL schema containing the table.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table that contains the column. This will be the SQL table name if one exists; otherwise, it will be the system table name.
COLUMN_NAME	NAME	VARCHAR(128)	Name of the column. This will be the SQL column name if one exists; otherwise, it will be the system column name.
ORDINAL_POSITION	COLNO	INTEGER	Numeric place of the column in the table, ordered from left to right.

Table 157. SYSFIELDS view (continued)

Column name	System Column Name	Data Type	Description
DATA_TYPE	COLTYPE	VARCHAR(8)	Type of column: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **FLOAT** Floating point; FLOAT, REAL, or DOUBLE PRECISION **DECFLOAT** Decimal floating-point **CHAR** Fixed-length character string **VARCHAR** Varying-length character string **CLOB** Character large object string **GRAPHIC** Fixed-length graphic string **VARG** Varying-length graphic string **DBCLOB** Double-byte character large object string **BINARY** Fixed-length binary string **VARBIN** Varying-length binary string **BLOB** Binary large object string **DATE** Date **TIME** Time **TIMESTMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **DISTINCT** Distinct type

Table 157. SYSFIELDS view (continued)

Column name	System Column Name	Data Type	Description
LENGTH	LENGTH	INTEGER	The length attribute of the column; or, in the case of a decimal, numeric, or nonzero precision binary column, its precision: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **Precision of number** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHAR **Maximum length of string** VARCHAR or CLOB **Length of graphic string** GRAPHIC **Maximum length of graphic string** VARGRAPHIC or DBCLOB **Length of string** BINARY **Maximum length of binary string** VARBIN or BLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **Maximum length of datalink URL and comment** DATALINK **40 bytes** ROWID **2147483647 bytes** XML **Same value as the source type** DISTINCT
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER Nullable	Maximum length of the string for binary, character, and graphic string and XML data types. Contains the null value if the column is not a string.
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER Nullable	Number of bytes for binary, character, and graphic string and XML data types. Contains the null value if the column is not a string.
NUMERIC_SCALE	SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if the column is not decimal, numeric, or binary.

Table 157. SYSFIELDS view (continued)

Column name	System Column Name	Data Type	Description
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	The precision of all numeric columns. **Note:** This column supplies the precision of all numeric data types, including decimal floating-point and single-and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the column is not numeric.
NUMERIC_PRECISION_RADIX	RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the column is not numeric.
CCSID	CCSID	INTEGER Nullable	The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB, XML, and DATALINK columns. Contains 65535 if the column is a BINARY, VARBIN, BLOB, or ROWID. Contains the null value if the column is a numeric data type.
DATETIME_PRECISION	DATPRC	INTEGER Nullable	The fractional part of a date, time, or timestamp. **0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if the column is not a date, time, or timestamp.
FIELD_PROC	FLDPROC	VARCHAR(279) Nullable	The name of the procedure.
PARMLIST	PARMLIST	DBCLOB(1M) CCSID 1200 Nullable	The parameter list following FIELDPROC in the statement that defined the field procedure with insignificant blanks removed.
EXITPARM	EXITPARM	BLOB(1M) Nullable	The parameter value block of the field procedure. This is the control block passed to the field procedure when it is invoked.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	The system name of the column
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	The system name of the table
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	The system name of the schema

SYSFUNCS

The SYSFUNCS view contains one row for each function created by the CREATE FUNCTION statement.

The following table describes the columns in the SYSFUNCS view:

Table 158. SYSFUNCS view

Column Name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSCHEMA	VARCHAR(128)	Schema name of the routine (function) instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
ROUTINE_SCHEMA	FUNCSCHEMA	VARCHAR(128)	Name of the SQL schema (schema) that contains the routine.
ROUTINE_NAME	FUNCNAME	VARCHAR(128)	Name of the routine.
ROUTINE_CREATED	RTNCREATE	TIMESTAMP	Identifies the timestamp when the routine was created.
ROUTINE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the routine.
ROUTINE_BODY	BODY	VARCHAR(8)	The type of the routine body: **EXTERNAL** This is an external routine. **SQL** This is an SQL routine.
EXTERNAL_NAME	EXTNAME	VARCHAR(279) Nullable	This column identifies the external program name. • For SQL functions or ILE service programs, the external program name is *schema-name/service-program-name(entry-point-name)*. • For Java programs, the external program name is an optional jar-id followed by a *fully-qualified-class-name!method-name* or *fully-qualified-class-name.method-name*. • For all other languages, the external program name is *schema-name/program-name*. Contains the null value if this is a system-generated function.
EXTERNAL_LANGUAGE	LANGUAGE	VARCHAR(8) Nullable	If this is an external routine, this column identifies the external program name. **C** The external program is written in C. **C++** The external program is written in C++. **CL** The external program is written in CL. **COBOL** The external program is written in COBOL. **COBOLLE** The external program is written in ILE COBOL. **JAVA** The external program is written in JAVA. **PLI** The external program is written in PL/I. **RPG** The external program is written in RPG. **RPGLE** The external program is written in ILE RPG. Contains the null value if this is not an external routine.

Table 158. SYSFUNCS view (continued)

Column Name	System Column Name	Data Type	Description
PARAMETER_STYLE	PARM_STYLE	VARCHAR(7) Nullable	If this is an external routine, this column identifies the parameter style (calling convention). **DB2SQL** This is the DB2SQL calling convention. **DB2GNRL** This is the DB2GENERAL calling convention. **GENERAL** This is the GENERAL calling convention. **JAVA** This is the JAVA calling convention. **NULLS** This is the GENERAL WITH NULLS calling convention. **SQL** This is the SQL standard calling convention. Contains the null value if this is not an external routine.
IS_DETERMINISTIC	DETERMINE	VARCHAR(3)	This column identifies whether the routine is deterministic. That is, whether a call to the routine with the same arguments will always return the same result. **NO** The routine is not deterministic. **YES** The routine is deterministic.
SQL_DATA_ACCESS	DATAACCESS	VARCHAR(8) Nullable	This column identifies whether a routine contains SQL and whether it reads or modifies data. **NONE** The routine does not contain any SQL statements. **CONTAINS** The routine contains SQL statements. **READS** The routine possibly reads data from a table or view. **MODIFIES** The routine possibly modifies data in a table or view or issues SQL DDL statements.
SQL_PATH	SQL_PATH	VARCHAR(3483) Nullable	If this is an SQL routine, this column identifies the path. Contains the null value if this is an external routine.
PARM_SIGNATURE	SIGNATURE	VARCHAR(2048)	This column identifies the routine signature.
NUMBER_OF_RESULTS	NUMRESULTS	SMALLINT Nullable	Identifies the number of results.
IN_PARMS	IN_PARMS	SMALLINT	Identifies the number of input parameters. 0 indicates that there are no input parameters.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.

Table 158. SYSFUNCS view (continued)

Column Name	System Column Name	Data Type	Description
ROUTINE_DEFINITION	ROUTINEDEF	DBCLOB(2M) CCSID 13488 Nullable	If this is an SQL routine, this column contains the SQL routine body. If this is an obfuscated routine, the text starts with the WRAPPED keyword and is followed by the encoded form of the statement text. Contains the null value if this is not an SQL routine.
FUNCTION_ORIGIN	ORIGIN	CHAR(1)	Identifies the type of function. If this is a procedure, this column contains a blank. **B** This is a built-in function (defined by DB2 for i). **E** This is a user-defined function. **U** This is a user-defined function that is based on another function. **S** This is a system-generated function.
FUNCTION_TYPE	TYPE	CHAR(1)	Identifies the form of the function. If this is a procedure, this column contains a blank. **S** This is a scalar function. **C** This is a column function. **T** This is a table function.
EXTERNAL_ACTION	EXT_ACTION	CHAR(1) Nullable	Identifies the whether the invocation of the function has external effects. **E** This function has external side effects. **N** This function does not have any external side effects.
IS_NULL_CALL	NULL_CALL	VARCHAR(3) Nullable	Identifies whether the function needs to be called if an input parameter is the null value. **NO** This function need not be called if an input parameter is the null value. If this is a scalar function, the result of the function is implicitly null if any of the operands are null. If this is a table function, the result of the function is an empty table if any of the operands are the null value. **YES** This function must be called even if an input operand is null.
SCRATCH_PAD	SCRATCHPAD	INTEGER Nullable	Identifies whether the address of a static memory area (scratch pad) is passed to the function. **0** The function does not have a scratch pad. **integer** Indicates the size of the scratch pad passed to the function.
FINAL_CALL	FINAL_CALL	VARCHAR(3) Nullable	Indicates whether a final call to the function should be made to allow the function to clean up its work areas (scratch pads). **NO** No final call is made. **YES** A final call to the function is made when the statement is complete.
PARALLELIZABLE	PARALLEL	VARCHAR(3) Nullable	Identifies whether the function can be run in parallel. **NO** The function must be synchronous. **YES** The function can be run in parallel.

Table 158. SYSFUNCS view (continued)

Column Name	System Column Name	Data Type	Description
DBINFO	DBINFO	VARCHAR(3) Nullable	Identifies whether information about the database is passed to the function. **NO** No database information is passed to the function. **YES** Information about the database is passed to the function.
SOURCE_ SPECIFIC_SCHEMA	SRCSCHEMA	VARCHAR(128) Nullable	If this is sourced function and the source is user-defined, this column contains the name of the source schema. If this is a sourced function and the source is built-in, this column contains 'QSYS2'. Contains the null value if this is not a sourced function.
SOURCE_SPECIFIC_NAME	SRCNAME	VARCHAR(128) Nullable	If this is sourced function and the source is user-defined, this column contains the specific name of the source function name. Contains the null value if this is not a sourced function.
IS_USER_DEFINED_CAST	CAST_FUNC	VARCHAR(3) Nullable	Identifies whether this function is a cast function created when a distinct type was created. **NO** This function is not a cast function. **YES** This function is a cast function.
CARDINALITY	CARD	BIGINT Nullable	Specifies the cardinality for a table function. Contains the null value if the function is not a table function or if cardinality was not specified.
FENCED	FENCED	VARCHAR(3) Nullable	Identifies whether the function is fenced. **NO** The function is not fenced. **YES** The function is fenced.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
ROUNDING_MODE	DECFLTRND	CHAR(1) Nullable	If this is an SQL function, identifies the DECFLOAT rounding mode. **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP Contains the null value if the function is not an SQL function.
INLINE	INLINE	VARCHAR(3) Nullable	Identifies whether the function can potentially be inlined. **NO** The function cannot be inlined. **YES** The function can be inlined. Contains the null value if the function is not an SQL function.
ROUTINE_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	Contains the label for a routine. Contains the null value if a label does not exist.

SYSINDEXES

The SYSINDEXES view contains one row for every index in the SQL schema created using the SQL CREATE INDEX statement, including indexes on the SQL catalog.

The following table describes the columns in the SYSINDEXES view:

Table 159. SYSINDEXES view

Column Name	System Column Name	Data Type	Description
INDEX_NAME	NAME	VARCHAR(128)	Name of the index. This will be the SQL index name if one exists; otherwise, it will be the system index name.
INDEX_OWNER	CREATOR	VARCHAR(128)	Owner of the index
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table on which the index is defined. This will be the SQL table name if one exists; otherwise, it will be the system table name.
TABLE_OWNER	TBCREATOR	VARCHAR(128)	Owner of the table
TABLE_SCHEMA	TBDBNAME	VARCHAR(128)	Name of the SQL schema that contains the table on which the index is defined
IS_UNIQUE	UNIQUERULE	CHAR(1)	If the index is unique: **D** No (duplicates are allowed) **V** Yes (duplicate NULL values are allowed) **U** Yes **E** Encoded vector index
COLUMN_COUNT	COLCOUNT	INTEGER	Number of columns in the key
INDEX_SCHEMA	DBNAME	VARCHAR(128)	Name of the SQL schema that contains the index
SYSTEM_INDEX_NAME	SYS_IXNAME	CHAR(10)	System index name
SYSTEM_INDEX_SCHEMA	SYS_IDNAME	CHAR(10)	System index schema name
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System table schema name
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
INDEX_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200	A character string supplied with the LABEL statement.
IS_SPANNING_INDEX	SPANNING	VARCHAR(3) Nullable	Indicates whether the index is built over multiple partitions or members: **DISTRIBUTED** The index is built over a distributed table. **NO** The index is not built over multiple partitions or members. **YES** The index is built over multiple partitions or members. Contains the null value if the base table is not a partitioned table.
INDEX_DEFINER	DEFINER	VARCHAR(128) Nullable	Name of the user that defined the index.

Table 159. SYSINDEXES view (continued)

Column Name	System Column Name	Data Type	Description
ROUNDING_MODE	DECFLTRND	CHAR(1) Nullable	Indicates the DECFLOAT rounding mode of the index: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP Contains the null value if the index does not have an expression that references a DECFLOAT column, function, or constant.
INDEX_HAS_SEARCH_CONDITION	IXHASWHERE	CHAR(1)	If the index has a search condition: **N** The index does not have a search condition. **Y** The index has a search condition.
SEARCH_CONDITION_HAS_UDF	IXWHEREUDF	CHAR(1)	If the index search condition contains a user-defined function: **N** The index is not sparse or does not contain a user-defined function. **Y** The index is sparse and the search condition contains a UDF.
SEARCH_CONDITION	IXWHERECON	DBCLOB(2M) CCSID 1200 Nullable	If the index is sparse, contains the search condition. Contains the null value if the index is not sparse.
INDEX_HAS_INCLUDE_EXPRESSION	IXHASINCEX	CHAR(1)	If the index contains an INCLUDE clause: **N** The index is does not contain an INCLUDE clause. **Y** The index contains an INCLUDE clause.
INCLUDE_EXPRESSION	IXINCEXPR	DBCLOB(2M) CCSID 1200 Nullable	If the index has an INCLUDE clause, contains the list of INCLUDE expressions. Contains the null value if there is no include clause.
CREATED	CREATED	TIMESTAMP	The timestamp when the SQL index was created.

SYSINDEXSTAT

The SYSINDEXSTAT view contains one row for every SQL index partition.

Use this view when you want to see information for a specific SQL index or set of SQL indexes. The information is similar to that returned via Show Indexes in System i Navigator.

The following table describes the columns in the SYSINDEXSTAT view:

Table 160. SYSINDEXSTAT view

Column name	System Column Name	Data Type	Description
INDEX_SCHEMA	INDSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the SQL index.
INDEX_NAME	INDNAME	VARCHAR(128)	Name of the SQL index.
INDEX_PARTITION	INDPART	VARCHAR(128)	Partition or member name of the SQL index.
INDEX_OWNER	INDOWNER	VARCHAR(128)	SQL index owner.
INDEX_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	Text of the SQL index. Contains null if text does not exist for the SQL index.
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
TABLE_PARTITION	TABPART	VARCHAR(128)	Name of the table partition or member.
INDEX_VALID	VALID	VARCHAR(3)	An indication or whether the SQL index is invalid and needs to be rebuilt: **NO** The SQL index is invalid. **YES** The SQL index is valid.
INDEX_CREATE_TIMESTAMP	INDCREATED	TIMESTAMP	The timestamp when the SQL index was created.
CREATE_TIMESTAMP	CREATED	TIMESTAMP	The timestamp when the SQL index partition was created.
LAST_BUILD_TIMESTAMP	LASTBUILD	TIMESTAMP Nullable	The timestamp when the SQL index was last rebuilt. Contains null if the SQL index has never been built.
LAST_QUERY_USE	LASTQRYUSE	TIMESTAMP Nullable	The timestamp of the last time the SQL index was used in a query since the last time the usage statistics were reset. If the SQL index has never been used in a query since the last time the usage statistics were reset, contains null.
LAST_STATISTICS_USE	LASTSTUSE	TIMESTAMP Nullable	The timestamp of the last time the SQL index was used by the optimizer for statistics since the last time the usage statistics were reset. If the SQL index has never been used for statistics since the last time the usage statistics were reset, contains null.
QUERY_USE_COUNT	QRYUSECNT	BIGINT	The number of times the SQL index was used in a query since the last time the usage statistics were reset. If the SQL index has never been used in a query since the last time the usage statistics were reset, contains 0.
QUERY_STATISTICS_COUNT	QRYSTCNT	BIGINT	The number of times the SQL index was used by the optimizer for statistics since the last time the usage statistics were reset. If the SQL index has never been used for statistics since the last time the usage statistics were reset, contains 0.
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP Nullable	The timestamp of the last time the SQL index was used directly by an application for native record I/O or SQL operations. If the SQL index has never been used, contains null.

Table 160. SYSINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
DAYS_USED_COUNT	DAYSUSED	INTEGER	The number of days the SQL index was used directly by an application for native record I/O or SQL operations since the last time the usage statistics were reset. If the SQL index has never been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	The timestamp of the last time the usage statistics were reset for the SQL index. For more information see the Change Object Description (CHGOBJD) command. If the SQL index's last used timestamp has never been reset, contains null.
NUMBER_KEY_COLUMNS	INDKEYS	BIGINT	Number of columns that define the SQL index key.
COLUMN_NAMES	COLNAMES	VARCHAR(1024)	A comma separated list of column names that define the SQL index key. If the length of all the column names exceeds 1024, '...' is returned at the end of the column value.
NUMBER_KEYS	NUMRIDS	BIGINT	Number of keys in the SQL index. If the SQL index is invalid, -1 is returned.
INDEX_SIZE	SIZE	BIGINT	Size (in bytes) of the binary tree or encoded vector index of the SQL index.
NUMBER_PAGES	PAGES	BIGINT Nullable	Number of pages in the SQL index. If the SQL index is invalid or is an encoded vector index, contains null.
LOGICAL_PAGE_SIZE	PAGE_SIZE	INTEGER Nullable	The logical page size of the index. If the index is an encoded vector index, contains null.
UNIQUE	UNIQUE	VARCHAR(21)	Indicates whether an SQL index is unique: **UNIQUE** The SQL index is a UNIQUE index. **UNIQUE WHERE NOT NULL** The SQL index is a UNIQUE WHERE NOT NULL index. **FIFO** The SQL index is a non-unique first-in-first-out (FIFO) index. **LIFO** The SQL index is a non-unique last-in-last-out (LIFO) index. **FCFO** The SQL index is a non-unique first-change-first-out (FCFO) index.
MAXIMUM_KEY_LENGTH	KEY_LENGTH	INTEGER Nullable	Maximum key length of an SQL index. If the SQL index is an encoded vector index, contains null.
UNIQUE_PARTIAL_KEY_VALUES	KEYCARDS	VARCHAR(96) Nullable	The unique partial key values for the SQL index. If the index is an encoded vector index, the first unique partial key value is the total number of unique values for the entire index key. The remaining unique partial key values returned are not applicable.
OVERFLOW_VALUES	OVERFLOW	INTEGER Nullable	The number of distinct key values that have overflowed the encoded vector index. If the SQL index is not an encoded vector index, contains null.
EVI_CODE_SIZE	CODE_SIZE	INTEGER Nullable	The size of the byte code of the encoded vector index. If the SQL index is not an encoded vector index, contains null.
SPARSE	SPARSE	VARCHAR(3)	Indicates whether the SQL index contains keys for all the rows of its depended on table: **NO** The index contains keys for all the rows of its depended on table. **YES** The SQL index includes a WHERE clause and does not contain keys for all the rows of its depended on table.

Table 160. SYSINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
DERIVED_KEY	DERIVED	VARCHAR(3)	Indicates whether the any key columns in the SQL index are expressions: **NO** No key columns of the SQL index are expressions. **YES** At least one key column is an expression.
PARTITIONED	PARTITION	VARCHAR(3)	Indicates whether the SQL index is partitioned or not partitioned: **DISTRIBUTED** The index is built over a distributed table. **NO** The SQL index is not partitioned (spans multiple partitions). **YES** The SQL index is not built over a partitioned table or built over a partitioned table and is partitioned (does not span multiple partitions). Contains the null value if the base table is not a partitioned table.
ACCPTH_TYPE	ACCPTHTYPE	VARCHAR(4)	Indicates the type of SQL index: **1 TB** The SQL index is a maximum 1 terabyte (*MAX1TB) binary radix index. **4 GB** The SQL index is a maximum 4 gigabyte (*MAX4GB) binary radix index. **EVI** The SQL index is an encoded vector index.
SORT_SEQUENCE	SRTSEQ	VARCHAR(12)	Indicates whether the SQL index uses a collating sequence: **BY HEX VALUE** The SQL index does not use a collating table. ***LANGIDSHR** The SQL index uses a shared weight collating sequence (SRTSEQ). ***LANGIDUNQ** The SQL index uses a unique weight collating sequence (SRTSEQ). **ALTSEQ** The SQL index uses an alternate collating sequence (ALTSEQ).
LANGUAGE_IDENTIFIER	LANGID	CHAR(3) Nullable	The language ID of the SQL index. Contains null if the collating sequence is hex.
SORT_SEQUENCE_SCHEMA	SRTSEQSCH	CHAR(10) Nullable	Schema name of the sort sequence to use. Contains null if there is no sort sequence schema name.
SORT_SEQUENCE_NAME	SRTSEQNAM	CHAR(10) Nullable	Name of the sort sequence to use. Contains null if there is no sort sequence name.
ESTIMATED_BUILD_TIME	ESTBLDTIME	INTEGER	Estimated time (in seconds) required to rebuild the SQL index.
LAST_BUILD_TIME	LSTBLDTIME	INTEGER Nullable	Elapsed time (in seconds) the last time the index was built. Contains null if the last build information is not available.

Table 160. SYSINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
LAST_BUILD_KEYS	LSTBLDKEYS	BIGINT Nullable	Number of keys the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_DEGREE	LSTBLDDEG	SMALLINT Nullable	Parallel degree the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_TYPE	LSTBLDTYPE	CHAR(1) Nullable	An indication of whether the last index build was a complete build or a build from the delayed maintenance keys: **0** The last rebuild of the index was from the delayed maintenance keys. **1** The last build or rebuild of the index was a complete build from the rows in the table. If the index has never been built, contains null.
LAST_INVALIDATION_TIMESTAMP	LSTINVAL	TIMESTAMP Nullable	An indication of when the index was last invalidated. If the index has never been invalidated, contains null.
INDEX_HELD	HELD	VARCHAR(3)	An indication of whether a pending rebuild of the SQL index is currently held by the user: **NO** A rebuild of the SQL index is not pending or is not held. **YES** A pending rebuild of the SQL index is held.
MAINTENANCE	MAINT	VARCHAR(11) Nullable	The maintenance of the SQL index: **REBUILD** The SQL index is not maintained and is rebuilt at open time. **DELAYED** The SQL index maintenance is delayed until the index is opened. **DO NOT WAIT** The SQL index is immediately maintained. If the SQL index is an encoded vector index, contains null.
DELAYED_MAINT_KEYS	DLYKEYS	INTEGER Nullable	Number of keys that need to be inserted into the binary tree of a delayed maintenance index. If the SQL index is not a delayed maintenance index, contains null.
RECOVERY	RECOVERY	VARCHAR(10) Nullable	The recovery attribute of the SQL index: **DURING IPL** The SQL index is recovered, if necessary, at IPL. **AFTER IPL** The SQL index is recovered, if necessary, after IPL. **NEXT OPEN** The SQL index is recovered, if necessary, on the next open. If the SQL index is an encoded vector index, contains null.

Table 160. SYSINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
ROUNDING_MODE	DECFLTRND	VARCHAR(8) Nullable	Indicates the DECFLOAT rounding mode of the index: **CEILING** ROUND_CEILING **DOWN** ROUND_DOWN **FLOOR** ROUND_FLOOR **HALFDOWN** ROUND_HALF_DOWN **HALFEVEN** ROUND_HALF_EVEN **HALFUP** ROUND_HALF_UP **UP** ROUND_UP Contains the null value if the index does not have an expression that references a DECFLOAT column, function, or constant.
DECFLOAT_WARNING	DECFLTWRN	VARCHAR(3) Nullable	Indicates whether DECFLOAT warnings are returned: **NO** DECFLOAT warnings are not returned. **YES** DECFLOAT warnings are returned. Contains the null value if the index does not have an expression that references a DECFLOAT column, function, or constant.
LOGICAL_READS	LGLREADS	BIGINT	Number of logical read operations for the SQL index since the last IPL.
SEQUENTIAL_READS	SEQREADS	BIGINT	Number of sequential read operations for the index since the last IPL.
RANDOM_READS	RANREADS	BIGINT	Number of random read operations for the index since the last IPL.
SEARCH_CONDITION	IXWHERECON	VARGRAPHIC(1024) CCSID 1200 Nullable	If an index is sparse, the search condition of the index. If the length of the search condition exceeds 1024, '...' is returned at the end of the column value. Contains null if the index is not sparse.
SEARCH_CONDITION_HAS_UDF	IXWHEREUDF	VARCHAR(3) Nullable	If an index is sparse, indicates whether the search condition of the index contains a user-defined function. Contains null if the index is not sparse. **NO** The index search condition does not contain a UDF. **YES** The index search condition contains a UDF.
KEEP_IN_MEMORY	KEEPINMEM	VARCHAR(3)	Indicates whether the index should be kept in memory: **NO** No memory preference. **YES** The index should be kept in memory, if possible.
MEDIA_PREFERENCE	MEDIAPREF	VARCHAR(3)	Indicates the media preference of the index: **ANY** No media preference. **SSD** The index should be allocated on Solid State Disk (SSD), if possible.
INCLUDE_EXPRESSION	IXINCEXPR	VARGRAPHIC(1024) CCSID 1200 Nullable	Index INCLUDE expression. Contains null if the index does not have an INCLUDE expression.
OWNING_INDEX_SCHEMA	OWNINDSCH	VARCHAR(128)	Name of the schema of the object that owns the index.
OWNING_INDEX_NAME	OWNINDNAME	VARCHAR(128)	Name of the object that owns the index.

Table 160. SYSINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
OWNING_INDEX_TYPE	OWNINDTYPE	VARCHAR(11)	The type of the object that owns the index: **INDEX** The owner is an SQL index. **LOGICAL** The owner is part of a logical file. **PHYSICAL** The owner is a part of a keyed physical file. **PRIMARY KEY** The owner is a primary key constraint. **UNIQUE** The owner is a unique constraint. **FOREIGN KEY** The owner is a foreign key constraint.
OWNING_INDEX_OWNER	OWNINDOWN	VARCHAR(128)	The owner of the object that owns the index.
OWNING_SYSTEM_INDEX_SCHEMA	OWNSYS_IXD	CHAR(10) Nullable	System index schema name of the owner of the index. Contains null if the owner is a constraint.
OWNING_SYSTEM_INDEX_NAME	OWNSYS_IXN	CHAR(10) Nullable	The system name of the owner of the index. Contains null if the owner is a constraint.
OWNING_INDEX_TEXT	OWNLABEL	VARGRAPHIC(50) CCSID 1200 Nullable	Text of the object that owns the index. Contains null if text does not exist for the object.
OWNING_INDEX_PARTITION	OWNINDMMBR	VARCHAR(128) Nullable	Partition or member name of the object that owns the index. Contains null if the owner is a constraint.
SYSTEM_INDEX_SCHEMA	SYS_IXDNAM	CHAR(10)	System index schema name.
SYSTEM_INDEX_NAME	SYS_IXNAME	CHAR(10)	System index name.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System table schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_MEMBER	SYS_MNAME	CHAR(10)	System member name.

SYSJARCONTENTS

The SYSJARCONTENTS table contains one row for each class defined by a jarid in the SQL schema.

The following table describes the columns in the SYSJARCONTENTS table.

Table 161. SYSJARCONTENTS table

Column Name	System Column Name	Data Type	Description
JARSCHEMA	JARSCHEMA	VARCHAR(128)	Name of the schema containing the jar_id.
JAR_ID	JAR_ID	VARCHAR(128)	Name of the jar_id.
CLASS	CLASS	VARCHAR(128)	Name of the class.
CLASS_SOURCE	CLASSSRC	DBCLOB(10485760) CCSID 13488 Nullable	Reserved. Contains the null value.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSJAROBJECTS

The SYSJAROBJECTS table contains one row for each jarid in the SQL schema.

The following table describes the columns in the SYSJAROBJECTS table.

Table 162. SYSJAROBJECTS table

Column Name	System Column Name	Data Type	Description
JARSCHEMA	JARSCHEMA	VARCHAR(128)	Name of the schema containing the jar_id.
JAR_ID	JAR_ID	VARCHAR(128)	Name of the jar_id.
DEFINER	DEFINER	VARCHAR(128)	Name of the owner of the jarid.
JAR_DATA	JAR_DATA	BLOB(104857600) Nullable	Byte-codes for the jar.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
JAR_CREATED	CREATEDTS	TIMESTAMP	Jar created timestamp
LAST_ALTERED	ALTEREDTS	TIMESTAMP Nullable	Reserved. Contains the null value.
DEBUG_MODE	DEBUG_MODE	CHAR(1)	Identifies whether the routine is debuggable. **0** The routine is not debuggable. **1** The routine is debuggable by the Unified Debugger. **2** The routine is debuggable by the system debugger. **N** The routine is disabled from being debugged by the Unified Debugger.
DEBUG_DATA	DEBUG_DATA	CLOB(1048576) Nullable	Reserved. Contains the null value.

SYSKEYCST

The SYSKEYCST view contains one or more rows for each UNIQUE KEY, PRIMARY KEY, or FOREIGN KEY in the SQL schema. There is one row for each column in every unique or primary key constraint and the referencing columns of a referential constraint.

The following table describes the columns in the SYSKEYCST view:

Table 163. SYSKEYCST view

Column Name	System Column Name	Data Type	Description
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
TABLE_SCHEMA	TDBNAME	VARCHAR(128)	Name of the schema containing the table.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table.
COLUMN_NAME	COLNAME	VARCHAR(128)	Name of the column.
ORDINAL_POSITION	COLSEQ	INTEGER	The position of the column within the key
COLUMN_POSITION	COLNO	INTEGER	The position of the column within the row
TABLE_OWNER	CREATOR	VARCHAR(128)	Owner of the table.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	System name of the column.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the schema table.
SYSTEM_CONSTRAINT_SCHEMA	SYS_CDNAME	CHAR(10)	Name of the system schema containing the constraint.

SYSKEYS

The SYSKEYS view contains one row for every column of an index in the SQL schema, including the keys for the indexes on the SQL catalog.

The following table describes the columns in the SYSKEYS view:

Table 164. SYSKEYS view

Column Name	System Column Name	Data Type	Description
INDEX_NAME	IXNAME	VARCHAR(128)	Name of the index. This will be the SQL index name if one exists; otherwise, it will be the system index name.
INDEX_OWNER	IXCREATOR	VARCHAR(128)	Owner of the index
COLUMN_NAME	COLNAME	VARCHAR(128)	Name of the column of the key. This will be the SQL column name if one exists; otherwise, it will be the system column name.
COLUMN_POSITION	COLNO	INTEGER	Numeric position of the column in the row.
		Nullable	Contains the null value if the key column is an expression.
ORDINAL_POSITION	COLSEQ	INTEGER	Numeric position of the column in the key
ORDERING	ORDERING	CHAR(1)	Order of the column in the key: **A** Ascending **D** Descending
INDEX_SCHEMA	IXDBNAME	VARCHAR(128)	Name of the schema containing the index.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	System name of the column
SYSTEM_INDEX_NAME	SYS_IXNAME	CHAR(10)	System name of the index
SYSTEM_INDEX_SCHEMA	SYS_IDNAME	CHAR(10)	System name of the schema containing the index
COLUMN_IS_EXPRESSION	IXISEXP	CHAR(1)	If the key column is an expression: **Y** The key column is not an expression. **N** The key column is an expression.
EXPRESSION_HAS_UDF	IXEXPUDF	CHAR(1)	If the key column is an expression and the expression contains a user-defined function:
		Nullable	**N** The key column is not an expression or the expression does not contain a user-defined function. **Y** The key column is an expression and the expression contains a UDF.
KEY_EXPRESSION	IXKEYEXP	DBCLOB(2097152)	If the key column is an expression, contains the expression.
		CCSID 1200 Nullable	Contains the column name if the key column is not an expression.

SYSMQTSTAT

The SYSMQTSTAT view contains one row for every materialized table partition.

Use this view when you want to see information about a specified materialized query table or set of materialized query tables. The information is similar to that returned via Show Materialized Query Tables in System i Navigator.

The following table describes the columns in the SYSMQTSTAT view:

Table 165. SYSMQTSTAT view

Column name	System Column Name	Data Type	Description
MQT_SCHEMA	MQTSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the materialized query table.
MQT_NAME	MQTNAME	VARCHAR(128)	Name of the materialized query table.
MQT_PARTITION	MQTMEMBER	VARCHAR(128)	Partition or member name of the materialized query table.
MQT_OWNER	MQTOWNER	VARCHAR(128)	Materialized query table owner.
MQT_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	Text of the materialized query table. Contains null if text does not exist for the materialized query table.
ENABLED	ENABLED	VARCHAR(3)	An indication of whether the materialized query table is enabled: **NO** The materialized query table is not enabled for use. **YES** The materialized query table is enabled for use by the database manager.
CREATE_TIMESTAMP	CREATED	TIMESTAMP	The timestamp when the materialized query table was created.
REFRESH_TIME	REFRESHDTS	TIMESTAMP Nullable	The timestamp when the materialized query table was last refreshed. Contains null if the materialized query table has never been refreshed.
LAST_QUERY_USE	LASTQRYUSE	TIMESTAMP Nullable	The timestamp of the last time the materialized query table was used in a query since the last time the usage statistics were reset. If the materialized query table has never been used in a query since the last time the usage statistics were reset, contains null.
LAST_STATISTICS_USE	LASTSTUSE	TIMESTAMP Nullable	The timestamp of the last time the materialized query table was used by the optimizer for statistics since the last time the usage statistics were reset. If the materialized query table has never been used for statistics since the last time the usage statistics were reset, contains null.
QUERY_USE_COUNT	QRYUSECNT	BIGINT	The number of times the materialized query table was used in a query since the last time the usage statistics were reset. If the materialized query table has never been used in a query since the last time the usage statistics were reset, contains 0.
QUERY_STATISTICS_COUNT	QRYSTCNT	BIGINT	The number of times the materialized query table was used by the optimizer for statistics since the last time the usage statistics were reset. If the materialized query table has never been used for statistics since the last time the usage statistics were reset, contains 0.
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP Nullable	The timestamp of the last time the materialized query table was used directly by an application for native record I/O or SQL operations. If the materialized query table has never been used, contains null.

Table 165. SYSMQTSTAT view (continued)

Column name	System Column Name	Data Type	Description
DAYS_USED_COUNT	DAYSUSED	INTEGER	The number of days the materialized query table was used directly by an application for native record I/O or SQL operations since the last time the usage statistics were reset. If the materialized query table has never been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	The timestamp of the last time the usage statistics were reset for the materialized query table. For more information see the Change Object Description (CHGOBJD) command. If the materialized query table's last used timestamp has never been reset, contains null.
NUMBER_ROWS	CARD	BIGINT	Number of rows in the materialized query table.
MQT_SIZE	SIZE	BIGINT	Size (in bytes) of the materialized query table.
LAST_CHANGE_TIMESTAMP	LASTCHG	TIMESTAMP Nullable	The timestamp of the last time the materialized query table was changed. If the materialized query table has never been changed since the last time the usage statistics were reset, contains null.
MAINTENANCE	MAINTAIN	VARCHAR(6)	Indicates the maintenance for the materialized query table: **SYSTEM** The materialized query table is system maintained. **USER** The materialized query table is user maintained.
INITIAL_DATA	INITIAL	VARCHAR(19)	Indicates the initial data for the materialized query table: **INITIALLY DEFERRED** Data is not inserted into the materialized query table when it is created. **INITIALLY IMMEDIATE** Data is inserted into the materialized query table when it is created.
REFRESH	REFRESH	VARCHAR(9)	Indicates when the data in the materialized query table can be refreshed: **DEFERRED** Data in the materialized query table can be refreshed at any time using the REFRESH TABLE statement. **IMMEDIATE** Data in the materialized query table is immediately refreshed.
ISOLATION	ISOLATION	VARCHAR(27)	Indicates the isolation level used to refresh the materialized query table: **NO COMMIT** The isolation level is NO COMMIT. **UNCOMMITTED READ** The isolation level is UNCOMMITTED READ. **CURSOR STABILITY** The isolation level is CURSOR STABILITY. **CURSOR STABILITY KEEP LOCKS** The isolation level is CURSOR STABILITY KEEP LOCKS. **READ STABILITY** The isolation level is READ STABILITY. **REPEATABLE READ** The isolation level is REPEATABLE READ.

Table 165. SYSMQTSTAT view (continued)

Column name	System Column Name	Data Type	Description
SORT_SEQUENCE	SRTSEQ	VARCHAR(12)	Indicates whether the materialize query table uses a collating sequence: **BY HEX VALUE** The materialize query table does not use a collating table. ***LANGIDSHR** The materialize query table uses a shared weight sort sequence (SRTSEQ). ***LANGIDUNQ** The materialize query table uses a unique weight sort sequence (SRTSEQ). **ALTSEQ** The materialize query table uses an alternate collating sequence (ALTSEQ).
LANGUAGE_IDENTIFIER	LANGID	CHAR(3) Nullable	The language ID of the materialize query table. Contains null if the sort sequence is hex.
SORT_SEQUENCE_SCHEMA	SRTSEQSCH	CHAR(10) Nullable	The sort sequence table system schema. Contains null if the sort sequence is hex.
SORT_SEQUENCE_NAME	SRTSEQNAME	CHAR(10) Nullable	The sort sequence table name. Contains null if the sort sequence is hex.
MQT_RESTORE_DEFERRED	MQTRSTDFR	VARCHAR(3)	An indication of whether a restore of the MQT is pending the restore of one of its dependents: **NO** The restore of the MQT is not deferred pending the restore of one of its dependent tables. **YES** The restore of the MQT is deferred pending the restore of one of its dependent tables.
ROUNDING_MODE	DECFLTRND	VARCHAR(8) Nullable	Indicates the DECFLOAT rounding mode of the materialized query table: **CEILING** ROUND_CEILING **DOWN** ROUND_DOWN **FLOOR** ROUND_FLOOR **HALFDOWN** ROUND_HALF_DOWN **HALFEVEN** ROUND_HALF_EVEN **HALFUP** ROUND_HALF_UP **UP** ROUND_UP Contains the null value if the materialized query table does not have an expression that references a DECFLOAT column, function, or constant.
DECFLOAT_WARNING	DECFLTWRN	VARCHAR(3) Nullable	Indicates whether DECFLOAT warnings are returned: **NO** DECFLOAT warnings are not returned. **YES** DECFLOAT warnings are returned. Contains the null value if the materialized query table does not have an expression that references a DECFLOAT column, function, or constant.
MQT_DEFINITION	MQTDEF	VARGRAPHIC(5000) CCSID 1200	The query of the materialized query table. If the length of the query exceeds 5000, '...' is returned at the end of the column value.

Table 165. SYSMQTSTAT view (continued)

Column name	System Column Name	Data Type	Description
SYSTEM_MQT_SCHEMA	SYS_MQDNAM	CHAR(10)	System materialized query table schema name.
SYSTEM_MQT_NAME	SYS_MQNAME	CHAR(10)	System materialized query table name.

SYSPACKAGE

The SYSPACKAGE view contains one row for each SQL package in the SQL schema.

The following table describes the columns in the SYSPACKAGE view:

Table 166. SYSPACKAGE view

Column Name	System Column Name	Data Type	Description
PACKAGE_CATALOG	LOCATION	VARCHAR(128)	Relational database name (RDBNAME) of the SQL package
PACKAGE_SCHEMA	COLLID	VARCHAR(128)	Name of the schema
PACKAGE_NAME	NAME	VARCHAR(128)	Name of the SQL package
PACKAGE_OWNER	OWNER	VARCHAR(128)	Owner of the SQL package
PACKAGE_CREATOR	CREATOR	VARCHAR(128)	Creator of the SQL package
CREATION_TIMESTAMP	TIMESTAMP	CHAR(26)	Timestamp of when the SQL package was created
DEFAULT_SCHEMA	QUALIFIER	VARCHAR(128)	Implicit name for unqualified tables, views, and indexes
PROGRAM_NAME	PROGNAME	VARCHAR(128)	Name of program the package was created from
PROGRAM_SCHEMA	LIBRARY	VARCHAR(128)	Name of schema containing the program
PROGRAM_CATALOG	RDB	VARCHAR(128)	Name of the relational database where the program resides
ISOLATION	ISOLATION	CHAR(2)	Isolation option specification: RR Repeatable Read (*RR) RS Read Stability (*ALL) CS Cursor Stability (*CS) UR Uncommitted Read (*CHG) NO None (*NONE)
QUOTE	QUOTE	CHAR(1)	Escape character specification (Y/N): Y = Quotation mark N = Apostrophe
COMMA	COMMA	CHAR(1)	Comma option specification (Y/N): Y = Comma N = Period
PACKAGE_TEXT	LABEL	VARCHAR(50)	A character string you supply with the LABEL statement.
LONG_COMMENT	REMARKS	VARCHAR(2000)	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
CONSISTENCY_TOKEN	CONTOKEN	CHAR(8) FOR BIT DATA	Consistency token of package
SYSTEM_PACKAGE_NAME	SYS_NAME	CHAR(10)	System name of the package.
SYSTEM_PACKAGE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the package.
SYSTEM_DEFAULT_SCHEMA	SYS_DDNAME	CHAR(10)	System name of the implicit qualifier for unqualified table, views, indexes, and packages.
SYSTEM_PROGRAM_NAME	SYS_PNAME	CHAR(10)	System name of the program.
SYSTEM_PROGRAM_SCHEMA	SYS_PDNAME	CHAR(10)	System name of the schema containing the program
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

Table 166. SYSPACKAGE view (continued)

Column Name	System Column Name	Data Type	Description
ROUNDING_MODE	DECFLTRND	CHAR(1)	The rounding mode for the package: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP
CONCURRENTACCESSRESOLUTION	CONCURRENT	CHAR(1)	Specifies the concurrent access resolution: **blank** Not specified **W** Wait for outcome **U** Use currently committed

SYSPACKAGEAUTH

The SYSPACKAGEAUTH view contains one row for every privilege granted on a package. Note that this catalog view cannot be used to determine whether a user is authorized to a package because the privilege to use a package could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the SYSPACKAGEAUTH view:

Table 167. SYSPACKAGEAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
PACKAGE_SCHEMA	COLLID	VARCHAR(128)	Name of the schema
PACKAGE_NAME	NAME	VARCHAR(128)	Name of the SQL package
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the package. **EXECUTE** The privilege to execute the package.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	If the privilege is granted through an authorization, contains the name of the authorization list. Contains the null value if the privilege is not granted through an authorization list.
SYSTEM_PACKAGE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the package
SYSTEM_PACKAGE_NAME	SYS_NAME	CHAR(10)	System name of the package

SYSPACKAGESTAT

The SYSPACKAGESTAT view contains one row for each SQL package in the SQL schema.

The following table describes the columns in the SYSPACKAGESTAT view:

Table 168. SYSPACKAGESTAT view

Column Name	System Column Name	Data Type	Description
PACKAGE_SCHEMA	COLLID	VARCHAR(128)	Name of the schema
PACKAGE_NAME	NAME	VARCHAR(128)	Name of the SQL package
PACKAGE_OWNER	OWNER	VARCHAR(128)	Owner of the SQL package
PACKAGE_CREATOR	CREATOR	VARCHAR(128)	Creator of the SQL package
CREATION_TIMESTAMP	TIMESTAMP	CHAR(26)	Timestamp of when the SQL package was created
DEFAULT_SCHEMA	QUALIFIER	VARCHAR(128)	Implicit name for unqualified tables, views, and indexes
ISOLATION	ISOLATION	CHAR(2)	Isolation option specification: **RR** Repeatable Read (*RR) **RS** Read Stability (*ALL) **CS** Cursor Stability (*CS) **UR** Uncommitted Read (*CHG) **NC** No Commit (*NONE)
CONCURRENTACCESSRESOLUTION	CONCURRENT	CHAR(1)	Specifies the concurrent access resolution: **blank** Not specified **W** Wait for outcome **U** Use currently committed
SECONDARY_SPACE_COUNT	PKG_SPACES	INTEGER	Number of spaces in the package.
PENDING_FULL	PENDFULL	VARCHAR(3) Nullable	Indicates whether the package is pending full. **NO** The package is not pending full. **YES** The package is pending full. Contains null for a DRDA package.
PACKAGE_TYPE	PKG_TYPE	VARCHAR(16)	Indicates the type of package. **EXTENDED DYNAMIC** The package is an extended dynamic package. **DRDA** The package is a DRDA package.
NUMBER_STATEMENTS	NBRSTMTS	INTEGER	Number of SQL statements in the package
PACKAGE_USED_SIZE	PKSIZE	INTEGER	Number of bytes that are used for SQL statements and access plans in the package.
NUMBER_DUMMIES	NBRDUMMIES	INTEGER Nullable	Number of dummy statements in the package. Contains null for a DRDA package.
NUMBER_COMPRESSIONS	PGM_CMP	INTEGER Nullable	Number of times the package has been compressed. Contains null for a DRDA package.
STATEMENT_CONTENTION_COUNT	CONTENTION	BIGINT	Number of times contention occurred when attempting to store a new access plan.
EXTENDED_INDICATOR	EXTIND	VARCHAR(9)	Indicates the EXTIND attribute: ***EXTIND** Extended indicator support is enabled. ***NOEXTIND** Extended indicator support is not enabled.

Table 168. SYSPACKAGESTAT view (continued)

Column Name	System Column Name	Data Type	Description
C_NUL_REQUIRED	CNULRQD	VARCHAR(10)	Indicates the CNULRQD attribute: ***CNULRQD** C nuls are required. ***NOCNULRQD** C nuls are not required.
NAMING	NAMING	VARCHAR(4)	Indicates the NAMING attribute: ***SYS** This is system naming. ***SQL** This is SQL naming.
TARGET_RELEASE	TGTRLS	VARCHAR(6)	Indicates the target release of the package (VxRxMx).
EARLIEST_POSSIBLE_RELEASE	MINRLS	VARCHAR(6) Nullable	Indicates the earliest IBM i release that supports all the SQL statements in the package (VxRxMx). **ANY** The statements are valid on any supported IBM i release. **VxRxMx** The statement is valid on IBM i VxRxMx release or later. Contains null if the earliest release has not yet been determined.
RDB	RDB	VARCHAR(18)	Indicates the RDB specified for the package. **rdb-name** The name of the relational database. ***NONE** A relational database was not specified.
CONSISTENCY_TOKEN	CONTOKEN	VARBINARY(8) Nullable	Indicates the consistency token of the package. Contains null if the package is not a DRDA package.
ALLOW_COPY_DATA	ALWCPYDTA	VARCHAR(9)	Indicates the ALWCPYDTA attribute: ***NO** A copy of the data is not allowed. ***OPTIMIZE** A copy of the data is allowed whenever it might result in better performance. ***YES** A copy of the data is allowed, but only when necessary.
LOB_FETCH_OPTIMIZATION	OPTLOB	VARCHAR(9)	Indicates the LOB optimization attribute: ***OPTLOB** The first FETCH for a cursor determines how the cursor will be used for LOB and XML result columns on all subsequent FETCHes. ***NOOPTLOB** Any FETCH may retrieve a LOB or XML result column into either a locator or variable.
DECIMAL_POINT	DECPNT	VARCHAR(7)	Indicates the decimal point for numeric constants used in SQL statements. ***PERIOD** The decimal point is a period. ***COMMA** The decimal point is a comma.
SQL_STRING_DELIMITER	STRDLM	VARCHAR(9)	Indicates the character used as the string delimiter in the SQL statements. ***APOSTSQL** The string delimiter is an apostrophe ('). ***QUOTESQL** The string delimiter is a quote (").

Table 168. SYSPACKAGESTAT view (continued)

Column Name	System Column Name	Data Type	Description
DATE_FORMAT	DATFMT	VARCHAR(4)	Indicates the DATFMT attribute: ***JOB** The date format specified in the job at runtime is used. ***USA** The date format is *USA. ***ISO** The date format is *ISO. ***EUR** The date format is *EUR. ***JIS** The date format is *JIS. ***MDY** The date format is *MDY. ***DMY** The date format is *DMY. ***YMD** The date format is *YMD. ***JUL** The date format is *JUL.
DATE_SEPARATOR	DATSEP	CHAR(1)	Indicates the date separator.
TIME_FORMAT	TIMFMT	VARCHAR(4)	Indicates the TIMFMT attribute: ***JOB** The time format specified in the job at runtime is used. ***USA** The time format is *USA. ***ISO** The time format is *ISO. ***EUR** The time format is *EUR. ***JIS** The time format is *JIS. ***HMS** The date format is *HMS.
TIME_SEPARATOR	TIMSEP	CHAR(1)	Indicates the time separator.
DYNAMIC_DEFAULT_SCHEMA	DYNDFTCOL	VARCHAR(4) Nullable	Indicates whether the value for DFTRDBCOL should be used for implicit qualification on dynamic SQL statements: ***NO** The schema specified in DFTDRBCOL is not used for dynamic SQL statements. ***YES** The schema specified in DFTDRBCOL is used for dynamic SQL statements. Contains null if a default schema was not specified (DFTRDBCOL).
CURRENT_RULES	SQLCURRULE	VARCHAR(4)	Indicates the SQLCURRULE attribute: ***DB2** The semantics of all SQL statements will default to the rules established for DB2. ***STD** The semantics of all SQL statements will default to the rules established by the ISO and ANSI SQL standards.

Table 168. SYSPACKAGESTAT view (continued)

Column Name	System Column Name	Data Type	Description
ALLOW_BLOCK	ALWBLK	VARCHAR(8)	Indicates the ALWBLK attribute: ***ALLREAD** Rows are blocked for read-only cursors. ***NONE** Rows are not blocked for retrieval of data for cursors. ***READ** Records are blocked for read-only retrieval of data for cursors when: • *NONE is specified for the Commitment control (COMMIT) parameter. • The cursor is declared with a FOR READ ONLY clause or there are no dynamic statements that could run a positioned UPDATE or DELETE statement for the cursor.
DELAY_PREPARE	DLYPRP	VARCHAR(4)	Indicates the DLYPRP attribute: ***NO** Dynamic statement validation is performed when the dynamic statements are prepared. ***YES** Dynamic statement validation is delayed until the dynamic statements are used.
USER_PROFILE	USRPRF	VARCHAR(7)	Specifies the user profile used for authority checking: ***USER** The profile of the user running statements in the package is used. ***OWNER** The profiles of both the owner of the package and the user running statements in the package is used. ***NAMING** If the naming convention is *SQL, *OWNER is used. If the naming convention is *SYS, *USER is used.
DYNAMIC_USER_PROFILE	DYNUSRPRF	VARCHAR(6)	Specifies the user profile used for dynamic SQL statements: ***USER** Local dynamic SQL statements are run under the profile of the job or thread. Distributed dynamic SQL statements are run under the profile of the application server job. ***OWNER** Local dynamic SQL statements are run under the profile of the package's owner. Distributed dynamic SQL statements are run under the profile of the SQL package's owner.

Table 168. SYSPACKAGESTAT view (continued)

Column Name	System Column Name	Data Type	Description
SORT_SEQUENCE	SRTSEQ	VARCHAR(12)	Indicates whether the package uses a collating sequence: **BY HEX VALUE** The package does not use a collating table. ***LANGIDSHR** The package uses a shared weight sort sequence (SRTSEQ). ***LANGIDUNQ** The package uses a unique weight sort sequence (SRTSEQ). **ALTSEQ** The package uses an alternate collating sequence (ALTSEQ).
LANGUAGE_IDENTIFIER	LANGID	CHAR(3) Nullable	The language ID sort sequence. Contains null if the sort sequence is not *LANGIDSHR or *LANGIDUNQ.
SORT_SEQUENCE_SCHEMA	SRTSEQSCH	CHAR(10) Nullable	The sort sequence table system schema. Contains null if the sort sequence is hex.
SORT_SEQUENCE_NAME	SRTSEQNAME	CHAR(10) Nullable	The sort sequence table name. Contains null if the sort sequence is hex.
RDB_CONNECTION_METHOD	RDBCNNMTH	VARCHAR(4)	Specifies the semantics used for CONNECT statements: ***RUW** CONNECT (Type 1) semantics are used to support remote unit of work. ***DUW** CONNECT (Type 2) semantics are used to support distributed unit of work.
DECRESULT_MAXIMUM_PRECISION	DECMAXPRC	SMALLINT	Specifies the maximum precision. **31** The maximum precision is 31. **63** The maximum precision is 63.
DECRESULT_MAXIMUM_SCALE	DECMAXSCL	SMALLINT	The maximum scale (number of decimal positions to the right of the decimal point) that should be returned for result data types.
DECRESULT_MINIMUM_DIVIDE_SCALE	DECMINDIV	SMALLINT	The minimum divide scale (number of decimal positions to the right of the decimal point) that should be returned for both intermediate and result data types.
DECFLOAT_ROUNDING_MODE	DECFLTRND	VARCHAR(8) Nullable	Indicates the DECFLOAT rounding mode: **CEILING** ROUND_CEILING **DOWN** ROUND_DOWN **FLOOR** ROUND_FLOOR **HALFDOWN** ROUND_HALF_DOWN **HALFEVEN** ROUND_HALF_EVEN **HALFUP** ROUND_HALF_UP **UP** ROUND_UP
DECFLOAT_WARNING	DECFLTWRN	VARCHAR(3)	Indicates whether DECFLOAT warnings are returned. **NO** DECFLOAT warnings are not returned. **YES** DECFLOAT warnings are returned.

Table 168. SYSPACKAGESTAT view (continued)

Column Name	System Column Name	Data Type	Description
SQLPATH	SQLPATH	VARCHAR(3483)	Identifies the SQL path.
		Nullable	Contains the null value if an SQL path is not specified.
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP	The timestamp of the last time the package was used. If the package has never been used, contains null.
		Nullable	
DAYS_USED_COUNT	DAYSUSED	INTEGER	The number of days the package was used since the last time the usage statistics were reset. If the package has never been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP	The timestamp of the last time the usage statistics were reset. If the statistics have never been reset, contains null.
		Nullable	
SYSTEM_PACKAGE_NAME	SYS_NAME	CHAR(10)	System name of the package.
SYSTEM_PACKAGE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the package.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSPACKAGESTMTSTAT

The SYSPACKAGESTMTSTAT view contains one row for each SQL statement in every SQL package.

The following table describes the columns in the SYSPACKAGESTMTSTAT view:

Table 169. SYSPACKAGESTMTSTAT view

Column Name	System Column Name	Data Type	Description
PACKAGE_SCHEMA	COLLID	VARCHAR(128)	Name of the schema.
PACKAGE_NAME	NAME	VARCHAR(128)	Name of the SQL package.
SPACE_NAME	SPCNAME	VARCHAR(10) Nullable	Generated space name within the package for extended dynamic packages. Contains the null value for DRDA packages which have only 1 internal space.
STATEMENT_NUMBER	STMTNBR	INTEGER	Number of this statement in the space.
STATEMENT_NAME	STMTNAME	VARCHAR(128) Nullable	Name of the statement. Contains the null value for DRDA packages.
NUMBER_TIMES_PREPARED	NBRPREP	INTEGER	Number of times this statement has been prepared.
NUMBER_TIMES_EXECUTED	NBREXEC	INTEGER	Number of times this statement has been executed.
ROWS_AFFECTED	ROWCNT	INTEGER	Total rows fetched, updated, inserted, or deleted for all executions of the statement. Contains 0 if the statement is not a FETCH, SET, VALUES INTO, UPDATE, INSERT, DELETE, or MERGE. Will contain 0 for a SET or VALUES INTO statement that is not implemented using a cursor.
COMPRESSES_SINCE_LAST_USED	CMPLU	INTEGER	Number of times the package has had storage compressed since this statement was last run. This value is used when processing the SQL_STMT_COMPRESS_MAX QAQQINI option to determine when to remove the space used for the QDT for a statement. Will always be 0 for DRDA packages.
PARAMETER_MARKER_CONVERTED	PMCNVT	CHAR(3)	For a dynamic statement, indicates whether literals were converted to parameter markers for statement reuse. **YES** Literals were converted to parameter markers. **NO** No conversion done or statement is not dynamic.
WITH_HOLD	WITHHOLD	CHAR(3) Nullable	Specifies the WITH HOLD option for statement: **YES** WITH HOLD clause specified. Contains the null value if the clause was not specified.
FETCH_ONLY	FETCHONLY	CHAR(3) Nullable	Specifies the FOR READ ONLY option for statement: **YES** FOR READ ONLY clause specified. Contains the null value if the clause was not specified.
CONCURRENTACCESSRESOLUTION	CONCURRENT	CHAR(1) Nullable	Specifies the concurrent access resolution for the statement: **W** Wait for outcome **U** Use currently committed **S** Skip locked data Contains the null value if concurrent access was not specified at the statement level.

Table 169. SYSPACKAGESTMTSTAT view (continued)

Column Name	System Column Name	Data Type	Description
NUMBER_REBUILDS	NBRREBLD	INTEGER	Number of times QDT or access plan has been rebuilt.
ISOLATION	ISOLATION	CHAR(2) Nullable	Isolation option specification: **RR** Repeatable Read (*RR) **RS** Read Stability (*ALL) **CS** Cursor Stability (*CS) **UR** Uncommitted Read (*CHG) **NC** No Commit (*NONE) Contains the null value if isolation level was not specified at the statement level.
NUMBER_ROWS_TO_OPTIMIZE	OPTROWS	INTEGER Nullable	Number of rows specified on the OPTIMIZE FOR *n* ROWS clause. -1 means that the value *ALL was specified. Contains the null value if the clause was not specified.
NUMBER_ROWS_TO_FETCH	FETCHROWS	INTEGER Nullable	Number of rows specified on FETCH FIRST *n* ROWS clause. Contains the null value if the clause was not specified.
LAST_QDT_REBUILD_REASON	QDTRBLD	CHAR(2) Nullable	Reason code for last QDT rebuild. This corresponds to column QVC22 in the STRDBMON outfile when QQRID = 1000. Contains the null value if the statement does not use a QDT or has never had a QDT rebuilt.
STATEMENT_TEXT	STMTTEXT	DBCLOB(2M) CCSID(1200)	Text of the SQL statement.
SYSTEM_PACKAGE_NAME	SYS_NAME	CHAR(10)	System name of the package.
SYSTEM_PACKAGE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the package.
ACCESS_PLAN_LENGTH	AP_LENGTH	INTEGER	Number of bytes that are used for the QDT and access plan for the statement.

SYSPARMS

The SYSPARMS table contains one row for each parameter of a procedure created by the CREATE PROCEDURE statement or function created by the CREATE FUNCTION statement. The result of a scalar function and the result columns of a table function are also returned.

The following table describes the columns in the SYSPARMS table:

Table 170. SYSPARMS table

Column Name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSCHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
ORDINAL_POSITION	PARMNO	INTEGER	Numeric place of the parameter in the parameter list, ordered from left to right from 1 (leftmost parameter) to *n* (*n*th parameter). For scalar functions, the result of the function has a value of *n*+1. For table functions, the result columns are numbered from *n*+1 (leftmost result column) to *n*+*m* (*m*th result column).
PARAMETER_MODE	PARMMODE	VARCHAR(5)	Type of the parameter: **IN** This is an input parameter. **OUT** This is an output parameter. **INOUT** This is an input/output parameter.
PARAMETER_NAME	PARMNAME	VARCHAR(128) Nullable	Name of the parameter. Contains the null value if the parameter does not have a name.

Table 170. SYSPARMS table (continued)

Column Name	System Column Name	Data Type	Description
DATA_TYPE	DATA_TYPE	VARCHAR(128)	Type of parameter: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **DOUBLE PRECISION** Floating point; DOUBLE PRECISION **REAL** Floating point; REAL **DECFLOAT** Decimal floating-point **CHARACTER** Fixed-length character string **CHARACTER VARYING** Varying-length character string **CHARACTER LARGE OBJECT** Character large object string **GRAPHIC** Fixed-length graphic string **GRAPHIC VARYING** Varying-length graphic string **DOUBLE-BYTE CHARACTER LARGE OBJECT** Double-byte character large object string **BINARY** Fixed-length binary string **BINARY VARYING** Varying-length binary string **BINARY LARGE OBJECT** Binary large object string **DATE** Date **TIME** Time **TIMESTAMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **DISTINCT** Distinct type **ARRAY** Array type
NUMERIC_SCALE	SCALE	INTEGER	Scale of numeric data.
		Nullable	Contains the null value if the parameter is not decimal, numeric, or binary.

Table 170. SYSPARMS table (continued)

Column Name	System Column Name	Data Type	Description
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	The precision of all numeric parameters. **Note:** This column supplies the precision of all numeric data types, including decimal floating-point and single-and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the parameter is not numeric.
CCSID	CCSID	INTEGER Nullable	The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB, and DATALINK parameters. A CCSID of 0 indicates that the CCSID of the job at run time is used. XML parameters use the value of SQL_XML_DATA_CCSID from the QAQQINI file. Contains the null value if the parameter is numeric.
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER Nullable	Maximum length of the string for binary, character, and graphic string and XML data types. Contains the null value if the parameter is not a string.
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER Nullable	Number of bytes for binary, character, and graphic string and XML data types. Contains the null value if the parameter is not a string.
NUMERIC_PRECISION_RADIX	RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the parameter is not numeric.
DATETIME_PRECISION	DATPRC	INTEGER Nullable	The fractional part of a date, time, or timestamp. **0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if the parameter is not date, time, or timestamp.
IS_NULLABLE	NULLS	VARCHAR(3)	Indicates whether the parameter is nullable. **NO** The parameter does not allow nulls. **YES** The parameter does allow nulls.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.

Table 170. SYSPARMS table (continued)

Column Name	System Column Name	Data Type	Description
ROW_TYPE	ROWTYPE	CHAR(1) Nullable	Indicates the type of row. **P** Parameter. **R** If the function is a table function, this indicates a result column. Otherwise, the result before casting. **C** Result after casting.
DATA_TYPE_SCHEMA	TYPESCHEMA	VARCHAR(128) Nullable	Schema of the data type if this is a distinct type. Contains the null value if the parameter is not a distinct type.
DATA_TYPE_NAME	TYPENAME	VARCHAR(128) Nullable	Name of the data type if this is a distinct type. Contains the null value if the parameter is not a distinct type.
AS_LOCATOR	ASLOCATOR	VARCHAR(3)	Indicates whether the parameter was specified as a locator. **NO** The parameter was not specified as a locator. **YES** The parameter was specified as a locator.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
NORMALIZE_DATA	NORMALIZE	VARCHAR(3) Nullable	Indicates whether the parameter value should be normalized or not. This attribute only applies to UTF-8 and UTF–16 data. **NO** The value should not be normalized. **YES** The value should be normalized.
DEFAULT	DEFAULT	DBCLOB(64K) CCSID 1200 Nullable	The expression string used to calculate the default value of a parameter, if one exists. If the default value is the null value, the expression string is the keyword NULL. Contains the null value if the parameter has no default.

SYSPARTITIONDISK

The SYSPARTITIONDISK view contains one row for every disk unit used to store data of every table partition or table member. If the table is a distributed table, the partitions that reside on other database nodes are not contained in this catalog view. They are contained in the catalog views of the other database nodes.

The following table describes the columns in the SYSPARTITIONDISK view:

Table 171. SYSPARTITIONDISK view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
TABLE_PARTITION	TABPART	VARCHAR(128)	Name of the table partition or member.
ASP_NUMBER	ASP_NUMBER	SMALLINT	Auxiliary Storage Pool (ASP) containing the partition.
DISK_TYPE	DISK_TYPE	CHAR(4)	Disk type number of the disk.
DISK_MODEL	DISK_MODEL	CHAR(4)	Model number of the disk.
UNIT_NUMBER	UNITNBR	SMALLINT	Unit number of the disk.
LOGICAL_MIRRORED_PAIR_STATUS	MIRRORPS	CHAR(1) Nullable	Indicates the status of a mirrored pair of disks: **0** Indicates that one mirrored unit of a mirrored pair is not active. **1** Indicates that both mirrored units of a mirrored pair are active. Contains null if the unit is not mirrored.
MIRRORED_UNIT_STATUS	MIRRORUS	CHAR(1) Nullable	Indicates the status of a mirrored unit: **1** Indicates that this mirrored unit of a mirrored pair is active (online with current data). **2** Indicates that this mirrored unit is being synchronized. **3** Indicates that this mirrored unit is suspended. Contains null if the unit is not mirrored.
UNIT_MEDIA_CAPACITY	UNITMCAP	BIGINT	Storage capacity (in bytes) of the unit.
UNIT_SPACE_AVAILABLE	UNITSPACE	BIGINT	Space (in bytes) available on the unit for use.
UNIT_SPACE_RESERVED_FOR_SYSTEM	UNITSRES	BIGINT	Space (in bytes) reserved on the unit for use by the system.
UNIT_SPACE_USED	UNITSUSED	BIGINT	Space (in bytes) on the unit used for the partition.
UNIT_TOTAL_ACCESS_TIME	UNITTATIME	INTEGER	The estimated time, in milliseconds, required to sequentially read the fixed-length data on the unit for the partition. The time is based on the amount of data on the unit and the average access time of the unit. The estimate assumes there will be no contention with other threads for the disk.
UNIT_TYPE	UNIT_TYPE	SMALLINT	Indicates the type of disk unit: **0** Not Solid State Disk (SSD). **1** Solid State Disk (SSD).
DATA_SEGMENT_TYPE	SEGMENT	SMALLINT	Segment type of the disk. **0** Indicates that this segment is for fixed-length data. **1** Indicates that this segment is for variable-length data.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.

Table 171. SYSPARTITIONDISK view (continued)

Column name	System Column Name	Data Type	Description
SYSTEM_TABLE_MEMBER	SYS_MNAME	CHAR(10)	System member name.

SYSPARTITIONINDEXDISK

The SYSPARTITIONINDEXDISK view contains one row for every disk unit used to store the index data of every table partition or table member. If the index is a distributed index, the partitions that reside on other database nodes are not contained in this catalog view. They are contained in the catalog views of the other database nodes.

The following table describes the columns in the SYSPARTITIONINDEXDISK view:

Table 172. SYSPARTITIONINDEXDISK view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
TABLE_PARTITION	TABPART	VARCHAR(128)	Name of the table partition or member.
PARTITION_TYPE	PARTTYPE	CHAR(1)	The type of the table partitioning: **blank** The table is not partitioned. **H** This is data hash partitioning. **R** This is data range partitioning. **D** This is distributed database hash partitioning.
PARTITION_NUMBER	PARTNBR	INTEGER Nullable	The partition number of this partition. If the table is a distributed table, contains null.
NUMBER_DISTRIBUTED_PARTITIONS	DSTPARTS	INTEGER Nullable	If the table is a distributed table, contains the total number of partitions. If the table is not a distributed table, contains null.
INDEX_SCHEMA	INDSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the index, logical file, or constraint.
INDEX_NAME	INDNAME	VARCHAR(128)	Name of the index, logical file, or constraint.
INDEX_MEMBER	INDMEMBER	VARCHAR(128) Nullable	Name of the member of the index or logical file. If the index type is a constraint, the member name is null.
INDEX_TYPE	INDTYPE	VARCHAR(11)	The type of the index: **INDEX** The index is an SQL index. **LOGICAL** The index is part of a logical file. **PHYSICAL** The index is part of a keyed physical file. **PRIMARY KEY** The index is a primary key constraint. **UNIQUE** The index is a unique constraint. **REFERENTIAL** The index is a foreign key constraint.
ASP_NUMBER	ASP_NUMBER	SMALLINT	Auxiliary Storage Pool (ASP) containing the index.
DISK_TYPE	DISK_TYPE	CHAR(4)	Disk type number of the disk.
DISK_MODEL	DISK_MODEL	CHAR(4)	Model number of the disk.
UNIT_NUMBER	UNITNBR	SMALLINT	Unit number of the disk.

Table 172. SYSPARTITIONINDEXDISK view (continued)

Column name	System Column Name	Data Type	Description
LOGICAL_MIRRORED_PAIR_STATUS	MIRRORPS	CHAR(1) Nullable	Indicates the status of a mirrored pair of disks: **0** Indicates that one mirrored unit of a mirrored pair is not active. **1** Indicates that both mirrored units of a mirrored pair are active. Contains null if the unit is not mirrored.
MIRRORED_UNIT_STATUS	MIRRORUS	CHAR(1) Nullable	Indicates the status of a mirrored unit: **1** Indicates that this mirrored unit of a mirrored pair is active (online with current data). **2** Indicates that this mirrored unit is being synchronized. **3** Indicates that this mirrored unit is suspended. Contains null if the unit is not mirrored.
UNIT_MEDIA_CAPACITY	UNITMCAP	BIGINT	Storage capacity (in bytes) of the unit.
UNIT_SPACE_AVAILABLE	UNITSPACE	BIGINT	Space (in bytes) available on the unit for use.
UNIT_SPACE_RESERVED_FOR_SYSTEM	UNITSRES	BIGINT	Space (in bytes) reserved on the unit for use by the system.
UNIT_SPACE_USED	UNITSUSED	BIGINT	Space (in bytes) on the unit used for the index.
UNIT_TYPE	UNIT_TYPE	SMALLINT	Indicates the type of disk unit: **0** Not Solid State Disk (SSD). **1** Solid State Disk (SSD).
SYSTEM_INDEX_SCHEMA	SYS_IXDNAME	CHAR(10)	System index schema name.
SYSTEM_INDEX_NAME	SYS_IXNAME	CHAR(10)	System index name.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_MEMBER	SYS_MNAME	CHAR(10)	System member name.

SYSPARTITIONINDEXES

The SYSPARTITIONINDEXES view contains one row for every index built over a table partition or table member. If the table is a distributed table, the indexes over partitions that reside on other database nodes are not contained in this catalog view. They are contained in the catalog views of the other database nodes.

Use this view when you want to see index information for indexes built on a specified table or set of tables. The information is similar to that returned via Show Indexes in System i Navigator.

The following table describes the columns in the SYSPARTITIONINDEXES view:

Table 173. SYSPARTITIONINDEXES view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
TABLE_PARTITION	TABPART	VARCHAR(128)	Name of the table partition or member.
INDEX_NAME	INDNAME	VARCHAR(128)	Name of the index, logical file, or constraint. If the index type indicates one or more temporary indexes, INDEX_NAME contains the number of maintained temporary indexes that currently exist on the table followed by the string 'MAINTAINED TEMPORARY INDEXES'.
INDEX_TYPE	INDTYPE	VARCHAR(11)	The type of the index: **INDEX** The index is an SQL index. **LOGICAL** The index is part of a logical file. **PHYSICAL** The index is a part of a keyed physical file. **PRIMARY KEY** The index is a primary key constraint. **UNIQUE** The index is a unique constraint. **REFERENTIAL** The index is a foreign key constraint. **TEMPORARY** Indicates one or more temporary indexes exist on the table.
INDEX_SCHEMA	INDSCHEMA	VARCHAR(128) Nullable	Name of the SQL schema that contains the index, logical file, or constraint. Contains null if the row indicates one or more maintained temporary indexes.
INDEX_OWNER	INDOWNER	VARCHAR(128) Nullable	Index owner. Contains null if the row indicates one or more maintained temporary indexes.
SYSTEM_INDEX_SCHEMA	SYS_IXDNAM	CHAR(10) Nullable	System index schema name. Contains null unless the index type is INDEX or LOGICAL.
SYSTEM_INDEX_NAME	SYS_IXNAME	CHAR(10) Nullable	System index name. Contains null unless the index type is INDEX or LOGICAL.
INDEX_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	Text of the index, logical file, or constraint. Contains null if text does not exist for the index.
INDEX_PARTITION	INDMEMBER	VARCHAR(128) Nullable	Partition or member name of the index. Contains null if the row indicates one or more maintained temporary indexes.

Table 173. SYSPARTITIONINDEXES view (continued)

Column name	System Column Name	Data Type	Description
INDEX_VALID	VALID	VARCHAR(3)	An indication or whether the index is invalid and needs to be rebuilt: **NO** The index is invalid. **YES** The index is valid.
CREATE_TIMESTAMP	CREATED	TIMESTAMP Nullable	The timestamp when the index was created. Contains null if the row indicates one or more maintained temporary indexes.
LAST_BUILD_TIMESTAMP	LASTBUILD	TIMESTAMP Nullable	The timestamp when the index was last rebuilt. Contains null if the row indicates one or more maintained temporary indexes.
LAST_QUERY_USE	LASTQRYUSE	TIMESTAMP Nullable	The timestamp of the last time the index was used in a query since the last time the usage statistics were reset. If the index has never been used in a query since the last time the usage statistics were reset or if the row indicates one or more maintained temporary indexes, contains null.
LAST_STATISTICS_USE	LASTSTUSE	TIMESTAMP Nullable	The timestamp of the last time the index was used by the optimizer for statistics since the last time the usage statistics were reset. If the index has never been used for statistics since the last time the usage statistics were reset or if the row indicates one or more maintained temporary indexes, contains null.
QUERY_USE_COUNT	QRYUSECNT	BIGINT	The number of times the index was used in a query since the last time the usage statistics were reset. If the index has never been used in a query since the last time the usage statistics were reset, contains 0.
QUERY_STATISTICS_COUNT	QRYSTCNT	BIGINT	The number of times the index was used by the optimizer for statistics since the last time the usage statistics were reset. If the index has never been used for statistics since the last time the usage statistics were reset, contains 0.
LAST_USED_TIMESTAMP		TIMESTAMP Nullable	The timestamp of the last time the index was used directly by an application for native record I/O or SQL operations. If the index has never been used or if the row indicates one or more maintained temporary indexes, contains null.
DAYS_USED_COUNT	DAYSUSED	INTEGER	The number of days the index was used directly by an application for native record I/O or SQL operations since the last time the usage statistics were reset. If the index has never been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	The timestamp of the last time the usage statistics were reset for the index. For more information see the Change Object Description (CHGOBJD) command. If the index's last used timestamp has never been reset, contains null.
NUMBER_KEY_COLUMNS	INDKEYS	BIGINT Nullable	Number of columns that define the index key. Contains null if the row indicates one or more maintained temporary indexes.
COLUMN_NAMES	COLNAMES	VARCHAR(1024) Nullable	A comma separated list of column names that define the index key. If the length of all the column names exceeds 1024, '...' is returned at the end of the column value. Contains null if the row indicates one or more maintained temporary indexes.
NUMBER_KEYS	NUMRIDS	BIGINT Nullable	Number of keys in the index. If the index is invalid or is an encoded vector index, -1 is returned. Contains null if the row indicates one or more maintained temporary indexes.
INDEX_SIZE	SIZE	BIGINT	Size (in bytes) of the binary tree or encoded vector index of the index.

Table 173. SYSPARTITIONINDEXES view (continued)

Column name	System Column Name	Data Type	Description
NUMBER_PAGES	PAGES	BIGINT Nullable	Number of pages in the index. If the index is invalid or is an encoded vector index, contains null.
LOGICAL_PAGE_SIZE	PAGE_SIZE	INTEGER Nullable	The logical page size of the index. If the index is an encoded vector index or if the row indicates one or more maintained temporary indexes, contains null.
UNIQUE	UNIQUE	VARCHAR(21) Nullable	Indicates whether an index is unique: **UNIQUE** The index is a UNIQUE index. **UNIQUE WHERE NOT NULL** The index is a UNIQUE WHERE NOT NULL index. **FIFO** The index is a non-unique first-in-first-out (FIFO) index. **LIFO** The index is a non-unique last-in-last-out (LIFO) index. **FCFO** The index is a non-unique first-change-first-out (FCFO) index. Contains null if the row indicates one or more maintained temporary indexes.
MAXIMUM_KEY_LENGTH	KEY_LENGTH	INTEGER Nullable	Maximum key length of an index. If the index is an encoded vector index, contains null.
UNIQUE_PARTIAL_KEY_VALUES	KEYCARDS	VARCHAR(96) Nullable	The unique partial key values for the index. If the index is an encoded vector index, the first unique partial key value is the total number of unique values for the entire index key. The remaining unique partial key values returned are not applicable. If the index is one or more maintained temporary indexes, contains null.
OVERFLOW_VALUES	OVERFLOW	INTEGER Nullable	The number of distinct key values that have overflowed the encoded vector index. If the index is not an encoded vector index, contains null.
EVI_CODE_SIZE	CODE_SIZE	INTEGER Nullable	The size of the byte code of the encoded vector index. If the index is not an encoded vector index, contains null.
SPARSE	SPARSE	VARCHAR(3) Nullable	Indicates whether the index contains keys for all the rows of its depended on table: **NO** The index contains keys for all the rows of its depended on table. **YES** The index is a select/omit logical file or an SQL index with a WHERE clause and does not contain keys for all the rows of its depended on table. Contains null if the row indicates one or more maintained temporary indexes.
DERIVED_KEY	DERIVED	VARCHAR(3) Nullable	Indicates whether the any key columns in the index are expressions: **NO** No key columns of the index are expressions. **YES** At least one key column is an expression. Contains null if the row indicates one or more maintained temporary indexes.

Table 173. SYSPARTITIONINDEXES view (continued)

Column name	System Column Name	Data Type	Description
PARTITIONED	PARTITION	VARCHAR(20) Nullable	Indicates whether the index is partitioned or not partitioned: **NO** An SQL index is not partitioned (spans multiple partitions). **YES** The index is not built over a partitioned table or built over a partitioned table and is partitioned (does not span multiple partitions or members). **MULTI-MEMBER LOGICAL** The index is a logical file built over multiple partitions or members. Contains null if the row indicates one or more maintained temporary indexes.
ACCPTH_TYPE	ACCPTHTYPE	VARCHAR(4) Nullable	Indicates the type of index: **1 TB** The index is a maximum 1 terabyte (*MAX1TB) binary radix index. **4 GB** The index is a maximum 4 gigabyte (*MAX4GB) binary radix index. **EVI** The index is an encoded vector index. Contains null if the row indicates one or more maintained temporary indexes.
SORT_SEQUENCE	SRTSEQ	VARCHAR(12) Nullable	Indicates whether the index uses a collating sequence: **BY HEX VALUE** The index does not use a collating table. ***LANGIDSHR** The index uses a shared weight sort sequence (SRTSEQ). ***LANGIDUNQ** The index uses a unique weight sort sequence (SRTSEQ). **ALTSEQ** The index uses an alternate collating sequence (ALTSEQ). Contains null if the row indicates one or more maintained temporary indexes.
LANGUAGE_IDENTIFIER	LANGID	CHAR(3) Nullable	The language ID of the index. Contains null if the sort sequence is hex or if the row indicates one or more maintained temporary indexes.
SORT_SEQUENCE_SCHEMA	SRTSEQSCH	CHAR(10) Nullable	Schema name of the sort sequence to use. Contains null if there is no schema name.
SORT_SEQUENCE_NAME	SRTSEQNAM	CHAR(10) Nullable	Name of the sort sequence to use. Contains null if there is no sort sequence name.
ESTIMATED_BUILD_TIME	ESTBLDTIME	INTEGER Nullable	Estimated time (in seconds) required to rebuild the index. Contains null if the row indicates one or more maintained temporary indexes.
LAST_BUILD_TIME	LSTBLDTIME	INTEGER Nullable	Elapsed time (in seconds) the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_KEYS	LSTBLDKEYS	BIGINT Nullable	Number of keys the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_DEGREE	LSTBLDDEG	SMALLINT Nullable	Parallel degree the last time the index was built. Contains null if the last build information is not available.

Table 173. SYSPARTITIONINDEXES view (continued)

Column name	System Column Name	Data Type	Description
LAST_BUILD_TYPE	LSTBLDTYPE	CHAR(1) Nullable	An indication of whether the last index build was a complete build or a build from the delayed maintenance keys: **0** The last rebuild of the index was from the delayed maintenance keys. **1** The last build or rebuild of the index was a complete build from the rows in the table. If the index has never been built, contains null.
LAST_INVALIDATION_TIMESTAMP	LSTINVAL	TIMESTAMP Nullable	An indication of when the index was last invalidated. If the index has never been invalidated, contains null.
INDEX_HELD	HELD	VARCHAR(3)	An indication or whether a pending rebuild of the index is currently held by the user: **NO** A rebuild of the index is not pending or is not held. **YES** A pending rebuild of the index is held.
MAINTENANCE	MAINT	VARCHAR(11) Nullable	The maintenance of the index: **REBUILD** The index is not maintained and is rebuilt at open time. **DELAYED** The index maintenance is delayed until the index is opened. **DO NOT WAIT** The index is immediately maintained. If the index is an encoded vector index or if the row indicates one or more maintained temporary indexes, contains null.
DELAYED_MAINT_KEYS	DLYKEYS	INTEGER Nullable	Number of keys that need to be inserted into the binary tree of a delayed maintenance index. If the index is not a delayed maintenance index, contains null.
RECOVERY	RECOVERY	VARCHAR(10) Nullable	The recovery attribute of the index: **DURING IPL** The index is recovered, if necessary, at IPL. **AFTER IPL** The index is recovered, if necessary, after IPL. **NEXT OPEN** The index is recovered, if necessary, on the next open. If the index is an encoded vector index or if the row indicates one or more maintained temporary indexes, contains null.

Table 173. SYSPARTITIONINDEXES view (continued)

Column name	System Column Name	Data Type	Description
ROUNDING_MODE	DECFLTRND	VARCHAR(8) Nullable	Indicates the DECFLOAT rounding mode of the index: **CEILING** ROUND_CEILING **DOWN** ROUND_DOWN **FLOOR** ROUND_FLOOR **HALFDOWN** ROUND_HALF_DOWN **HALFEVEN** ROUND_HALF_EVEN **HALFUP** ROUND_HALF_UP **UP** ROUND_UP Contains the null value if the index does not have an expression that references a DECFLOAT column, function, or constant; or if the row indicates one or more maintained temporary indexes.
DECFLOAT_WARNING	DECFLTWRN	VARCHAR(3) Nullable	Indicates whether DECFLOAT warnings are returned: **NO** DECFLOAT warnings are not returned. **YES** DECFLOAT warnings are returned. Contains the null value if the index does not have an expression that references a DECFLOAT column, function, or constant; or if the row indicates one or more maintained temporary indexes.
LOGICAL_READS	LGLREADS	BIGINT Nullable	Number of logical read operations for the index since the last IPL. Contains null if the row indicates one or more maintained temporary indexes.
SEQUENTIAL_READS	SEQREADS	BIGINT	Number of sequential read operations for the index since the last IPL.
RANDOM_READS	RANREADS	BIGINT	Number of random read operations for the index since the last IPL.
SEARCH_CONDITION	IXWHERECON	VARGRAPHIC(1024) CCSID 1200 Nullable	If an index is sparse, the search condition of the index. If the length of the search condition exceeds 1024, '...' is returned at the end of the column value. Contains null if the index is not sparse.
SEARCH_CONDITION_HAS_UDF	IXWHEREUDF	VARCHAR(3) Nullable	If an index is sparse, indicates whether the search condition of the index contains a user-defined function. Contains null if the index is not sparse. **NO** The index search condition does not contain a UDF. **YES** The index search condition contains a UDF.
KEEP_IN_MEMORY	KEEPINMEM	VARCHAR(3)	Indicates whether the index should be kept in memory: **NO** No memory preference. **YES** The index should be kept in memory, if possible.
MEDIA_PREFERENCE	MEDIAPREF	VARCHAR(3)	Indicates the media preference of the index: **ANY** No media preference. **SSD** The index should be allocated on Solid State Disk (SSD), if possible.

Table 173. SYSPARTITIONINDEXES view (continued)

Column name	System Column Name	Data Type	Description
INCLUDE_EXPRESSION	IXINCEXPR	VARGRAPHIC(1024) CCSID 1200 Nullable	Index INCLUDE expression. Contains null if the index does not have an INCLUDE expression.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_MEMBER	SYS_MNAME	CHAR(10)	System member name.

SYSPARTITIONINDEXSTAT

The SYSPARTITIONINDEXSTAT view contains one row for every index built over a table partition or table member. Indexes that share another index's binary tree are not included. If the table is a distributed table, the indexes over partitions that reside on other database nodes are not contained in this catalog view. They are contained in the catalog views of the other database nodes.

The following table describes the columns in the SYSPARTITIONINDEXSTAT view:

Table 174. SYSPARTITIONINDEXSTAT view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
TABLE_PARTITION	TABPART	VARCHAR(128)	Name of the table partition or member.
PARTITION_TYPE	PARTTYPE	CHAR(1)	The type of the table partitioning: **blank** The table is not partitioned. **H** This is data hash partitioning. **R** This is data range partitioning. **D** This is distributed database hash partitioning.
PARTITION_NUMBER	PARTNBR	INTEGER Nullable	The partition number of this partition. If the table is a distributed table, contains null.
NUMBER_DISTRIBUTED_PARTITIONS	DSTPARTS	INTEGER Nullable	If the table is a distributed table, contains the total number of partitions. If the table is not a distributed table, contains null.
INDEX_SCHEMA	INDSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the index, logical file, or constraint.
INDEX_NAME	INDNAME	VARCHAR(128)	Name of the index, logical file, or constraint.
INDEX_MEMBER	INDMEMBER	VARCHAR(128) Nullable	Name of the member of the index or logical file. If the index type is a constraint, the member name is null.
INDEX_TYPE	INDTYPE	VARCHAR(11)	The type of the index: **INDEX** The index is an SQL index. **LOGICAL** The index is part of a logical file. **PHYSICAL** The index is part of a keyed physical file. **PRIMARY KEY** The index is a primary key constraint. **UNIQUE** The index is a unique constraint. **REFERENTIAL** The index is a foreign key constraint.
NUMBER_KEY_COLUMNS	INDKEYS	BIGINT	Number of columns that define the index key.
COLUMN_NAMES	COLNAMES	VARCHAR(1024)	A comma separated list of column names that define the index key. If the length of all the column names exceeds 1024, '...' is returned at the end of the column value.
NUMBER_LEAF_PAGES	NLEAF	BIGINT	Not applicable for DB2 for i. Will always be -1.
NUMBER_LEVELS	NLEVELS	SMALLINT	Not applicable for DB2 for i. Will always be -1.
FIRSTKEYCARD	KEYCARD1	BIGINT	Number of distinct first key values. If the index is an encoded vector index, this is the total number of unique values for the entire index key.

Table 174. SYSPARTITIONINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
FIRST2KEYCARD	KEYCARD2	BIGINT	Number of distinct keys using the first two columns of the index. If the index is an encoded vector index, -1 is returned.
FIRST3KEYCARD	KEYCARD3	BIGINT	Number of distinct keys using the first three columns of the index. If the index is an encoded vector index, -1 is returned.
FIRST4KEYCARD	KEYCARD4	BIGINT	Number of distinct keys using the first four columns of the index. If the index is an encoded vector index, -1 is returned.
FULLKEYCARD	KEYCARDF	BIGINT	Number of distinct full key values. If the index has more than 4 key columns or is an encoded vector index, -1 is returned.
CLUSTERRATIO	CLSRATIO	SMALLINT	Not applicable for DB2 for i. Will always be -1.
CLUSTERFACTOR	CLSFACTOR	DOUBLE	Not applicable for DB2 for i. Will always be -1.
SEQUENTIAL_PAGES	SEQPAGES	BIGINT	Not applicable for DB2 for i. Will always be -1.
DENSITY	DENSITY	INTEGER	Not applicable for DB2 for i. Will always be -1.
PAGE_FETCH_PAIRS	FETCHPAIRS	VARCHAR(520)	Not applicable for DB2 for i. Will always be an empty string.
NUMBER_KEYS	NUMRIDS	BIGINT	Number of keys in the index. If the index is invalid or is an encoded vector index, -1 is returned.
NUMRIDS_DELETED	NUMRIDSDLT	BIGINT	Not applicable for DB2 for i. Will always be 0.
NUM_EMPTY_LEAFS	EMPTYLEAFS	BIGINT	Not applicable for DB2 for i. Will always be 0.
AVERAGE_RANDOM_FETCH_PAGES	AVGRNDFTCH	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_RANDOM_PAGES	AVGRNDPAGE	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_SEQUENCE_GAP	AVGSEQGAP	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_SEQUENCE_FETCH_GAP	AVGSEQFGAP	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_SEQUENCE_PAGES	AVGSEQPAGE	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_SEQUENCE_FETCH_PAGES	AVGSEQFPAG	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVGPARTITION_CLUSTERRATIO	PCLSRATIO	SMALLINT	Not applicable for DB2 for i. Will always be -1.
AVGPARTITION_CLUSTERFACTOR	PCLSFACTOR	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVGPARTITION_PAGE_FETCH_PAIRS	PFETCHPAIR	VARCHAR(520)	Not applicable for DB2 for i. Will always be an empty string.
DATAPARTITION_CLUSTERFACTOR	DCLSFACTOR	DOUBLE	A statistic measuring the "clustering" of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.
INDCARD	INDCARD	BIGINT	Number of keys in the index. If the index is invalid or is an encoded vector index, -1 is returned.
INDEX_VALID	VALID	CHAR(1)	An indication or whether the index is invalid and needs to be rebuilt: **0** The index is invalid. **1** The index is valid.
INDEX_HELD	HELD	CHAR(1)	An indication or whether a pending rebuild of the index is currently held by the user: **0** A rebuild of the index is not pending or is not held. **1** A pending rebuild of the index is held.
CREATE_TIMESTAMP	CREATED	TIMESTAMP	The timestamp when the index was created.
LAST_BUILD_TIMESTAMP	LASTBUILD	TIMESTAMP	The timestamp when the index was last rebuilt.
LAST_QUERY_USE	LASTQRYUSE	TIMESTAMP Nullable	The timestamp of the last time the index was used in a query since the last time the usage statistics were reset. If the index has never been used in a query since the last time the usage statistics were reset, contains null.

Table 174. SYSPARTITIONINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
LAST_STATISTICS_USE	LASTSTUSE	TIMESTAMP Nullable	The timestamp of the last time the index was used by the optimizer for statistics since the last time the usage statistics were reset. If the index has never been used for statistics since the last time the usage statistics were reset, contains null.
QUERY_USE_COUNT	QRYUSECNT	BIGINT	The number of times the index was used in a query since the last time the usage statistics were reset. If the index has never been used in a query since the last time the usage statistics were reset, contains 0.
QUERY_STATISTICS_COUNT	QRYSTCNT	BIGINT	The number of times the index was used by the optimizer for statistics since the last time the usage statistics were reset. If the index has never been used for statistics since the last time the usage statistics were reset, contains 0.
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP Nullable	The timestamp of the last time the index was used directly by an application for native record I/O or SQL operations. If the index has never been used, contains null.
DAYS_USED_COUNT	DAYSUSED	INTEGER	The number of days the index was used directly by an application for native record I/O or SQL operations since the last time the usage statistics were reset. If the index has never been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	The timestamp of the last time the usage statistics were reset for the index. For more information see the Change Object Description (CHGOBJD) command. If the index's last used timestamp has never been reset, contains null.
INDEX_SIZE	SIZE	BIGINT	Size (in bytes) of the binary tree or encoded vector index of the index.
ESTIMATED_BUILD_TIME	ESTBLDTIME	INTEGER	Estimated time (in seconds) required to rebuild the index.
LAST_BUILD_TIME	LSTBLDTIME	INTEGER Nullable	Elapsed time (in seconds) the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_KEYS	LSTBLDKEYS	BIGINT Nullable	Number of keys the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_DEGREE	LSTBLDDEG	SMALLINT Nullable	Parallel degree the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_TYPE	LSTBLDTYPE	CHAR(1) Nullable	An indication of whether the last index build was a complete build or a build from the delayed maintenance keys: **0** The last rebuild of the index was from the delayed maintenance keys. **1** The last build or rebuild of the index was a complete build from the rows in the table. If the index has never been built, contains null.
LAST_INVALIDATION_TIMESTAMP	LSTINVAL	TIMESTAMP Nullable	An indication of when the index was last invalidated. If the index has never been invalidated, contains null.
DELAYED_MAINT_KEYS	DLYKEYS	INTEGER Nullable	Number of keys that need to be inserted into the binary tree of a delayed maintenance index. If the index is not a delayed maintenance index, contains null.

Table 174. SYSPARTITIONINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
SPARSE	SPARSE	CHAR(1)	Indicates whether the index contains keys for all the rows of its depended on table: **0** The index contains keys for all the rows of its depended on table. **1** The index is a select/omit logical file or SQL index with a WHERE clause and does not contain keys for all the rows of its depended on table.
DERIVED_KEY	DERIVED	CHAR(1)	Indicates whether the any key columns in the index are expressions: **0** No key columns of the index are expressions. **1** At least one key column is an expression. Currently, this is only possible in a DDS-created logical file or temporary index.
PARTITIONED	PARTITION	CHAR(1)	Indicates whether the index is partitioned or not partitioned: **0** An SQL index is not partitioned (spans multiple partitions). **1** The index is not built over a partitioned table or built over a partitioned table and is partitioned (does not span multiple partitions or members). **2** The index is a logical file built over multiple partitions or members.
ACCPTH_TYPE	ACCPTHTYPE	CHAR(1)	Indicates the type of index: **0** The index is a maximum 1 terabyte (*MAX1TB) binary radix index. **1** The index is a maximum 4 gigabyte (*MAX4GB) binary radix index. **2** The index is an encoded vector index.
UNIQUE	UNIQUE	CHAR(1)	Indicates whether an index is unique: **0** The index is a UNIQUE index. **1** The index is a UNIQUE WHERE NOT NULL index. **2** The index is a non-unique first-in-first-out (FIFO) index. **3** The index is a non-unique last-in-last-out (LIFO) index. **4** The index is a non-unique first-change-first-out (FCFO) index.
SRTSEQ_TYPE	SRTSEQ	CHAR(1)	Indicates whether the index uses a collating sequence: **0** The index does not use a collating table. **1** The index uses an alternate collating sequence (ALTSEQ). **2** The index uses a sort sequence (SRTSEQ).
LOGICAL_PAGE_SIZE	PAGE_SIZE	INTEGER Nullable	The logical page size of the index. If the index is an encoded vector index, contains null.

Table 174. SYSPARTITIONINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
OVERFLOW_VALUES	OVERFLOW	INTEGER Nullable	The number of distinct key values that have overflowed the encoded vector index. If the index is not an encoded vector index, contains null.
EVI_CODE_SIZE	CODE_SIZE	INTEGER Nullable	The size of the byte code of the encoded vector index. If the index is not an encoded vector index, contains null.
LOGICAL_READS	LGLREADS	BIGINT	Number of logical read operations for the index since the last IPL.
PHYSICAL_READS	PHYREADS	BIGINT	Not applicable for DB2 for i. Will always be 0.
SEQUENTIAL_READS	SEQREADS	BIGINT	Number of sequential read operations for the index since the last IPL.
RANDOM_READS	RANREADS	BIGINT	Number of random read operations for the index since the last IPL.
SEARCH_CONDITION	IXWHERECON	VARGRAPHIC(1024) CCSID 1200	If an index is sparse, the search condition of the index. If the length of the search condition exceeds 1024, '...' is returned at the end of the column value.
KEEP_IN_MEMORY	KEEPINMEM	CHAR(1)	Indicates whether the index should be kept in memory: **0** No memory preference. **1** The index should be kept in memory, if possible.
MEDIA_PREFERENCE	MEDIAPREF	SMALLINT	Indicates the media preference of the index: **0** No media preference. **255** The index should be allocated on Solid State Disk (SSD), if possible.
INCLUDE_EXPRESSION	IXINCEXPR	VARGRAPHIC(1024) CCSID 1200 Nullable	Index INCLUDE expression. Contains null if the index does not have an INCLUDE expression.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_MEMBER	SYS_MNAME	CHAR(10)	System member name.

SYSPARTITIONMQTS

The SYSPARTITIONMQTS view contains one row for every materialized table built over a table partition or table member. If the table is a distributed table, the materialized tables over partitions that reside on other database nodes are not contained in this catalog view. They are contained in the catalog views of the other database nodes.

Use this view when you want to see materialized query table information for materialized tables built on a specified table or set of tables. The information is similar to that returned via Show Materialized Query Tables in System i Navigator.

The following table describes the columns in the SYSPARTITIONMQTS view:

Table 175. SYSPARTITIONMQTS view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
TABLE_PARTITION	TABPART	VARCHAR(128)	Name of the table partition or member.
MQT_NAME	MQTNAME	VARCHAR(128)	Name of the materialized query table.
MQT_SCHEMA	MQTSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the materialized query table.
MQT_PARTITION	MQTMEMBER	VARCHAR(128)	Partition or member name of the materialized query table.
MQT_OWNER	MQTOWNER	VARCHAR(128)	Materialized query table owner.
SYSTEM_MQT_SCHEMA	SYS_MQDNAM	CHAR(10)	System materialized query table schema name.
SYSTEM_MQT_NAME	SYS_MQNAME	CHAR(10)	System materialized query table name.
ENABLED	ENABLED	VARCHAR(3)	An indication or whether the materialized query table is enabled: **NO** The materialized query table is not enabled for use. **YES** The materialized query table is enabled for use by the database manager.
CREATE_TIMESTAMP	CREATED	TIMESTAMP	The timestamp when the materialized query table was created.
REFRESH_TIME	REFRESHDTS	TIMESTAMP Nullable	The timestamp when the materialized query table was last refreshed. Contains null if the materialized query table has never been refreshed.
LAST_QUERY_USE	LASTQRYUSE	TIMESTAMP Nullable	The timestamp of the last time the materialized query table was used in a query since the last time the usage statistics were reset. If the materialized query table has never been used in a query since the last time the usage statistics were reset, contains null.
LAST_STATISTICS_USE	LASTSTUSE	TIMESTAMP Nullable	The timestamp of the last time the materialized query table was used by the optimizer for statistics since the last time the usage statistics were reset. If the materialized query table has never been used for statistics since the last time the usage statistics were reset, contains null.
QUERY_USE_COUNT	QRYUSECNT	BIGINT	The number of times the materialized query table was used in a query since the last time the usage statistics were reset. If the materialized query table has never been used in a query since the last time the usage statistics were reset, contains 0.
QUERY_STATISTICS_COUNT	QRYSTCNT	BIGINT	The number of times the materialized query table was used by the optimizer for statistics since the last time the usage statistics were reset. If the materialized query table has never been used for statistics since the last time the usage statistics were reset, contains 0.

Table 175. SYSPARTITIONMQTS view (continued)

Column name	System Column Name	Data Type	Description
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP Nullable	The timestamp of the last time the materialized query table was used directly by an application for native record I/O or SQL operations. If the materialized query table has never been used, contains null.
DAYS_USED_COUNT	DAYSUSED	INTEGER	The number of days the materialized query table was used directly by an application for native record I/O or SQL operations since the last time the usage statistics were reset. If the materialized query table has never been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	The timestamp of the last time the usage statistics were reset for the materialized query table. For more information see the Change Object Description (CHGOBJD) command. If the materialized query table's last used timestamp has never been reset, contains null.
NUMBER_ROWS	CARD	BIGINT	Number of rows in the materialized query table.
MQT_SIZE	SIZE	BIGINT	Size (in bytes) of the materialized query table.
LAST_CHANGE_TIMESTAMP	LASTCHG	TIMESTAMP Nullable	The timestamp of the last time the materialized query table was changed. If the materialized query table has never been changed since the last time the usage statistics were reset, contains null.
MAINTENANCE	MAINTAIN	VARCHAR(6)	Indicates the maintenance for the materialized query table: **SYSTEM** The materialized query table is system maintained. **USER** The materialized query table is user maintained.
INITIAL_DATA	INITIAL	VARCHAR(19)	Indicates the initial data for the materialized query table: **INITIALLY DEFERRED** Data is not inserted into the materialized query table when it is created. **INITIALLY IMMEDIATE** Data is inserted into the materialized query table when it is created.
REFRESH	REFRESH	VARCHAR(9)	Indicates when the data in the materialized query table can be refreshed: **DEFERRED** Data in the materialized query table can be refreshed at any time using the REFRESH TABLE statement. **IMMEDIATE** Data in the materialized query table is immediately refreshed.

Table 175. SYSPARTITIONMQTS view (continued)

Column name	System Column Name	Data Type	Description
ISOLATION	ISOLATION	VARCHAR(27)	Indicates the isolation level used to refresh the materialized query table: **NO COMMIT** The isolation level is NO COMMIT. **UNCOMMITTED READ** The isolation level is UNCOMMITTED READ. **CURSOR STABILITY** The isolation level is CURSOR STABILITY. **CURSOR STABILITY KEEP LOCKS** The isolation level is CURSOR STABILITY KEEP LOCKS. **READ STABILITY** The isolation level is READ STABILITY. **REPEATABLE READ** The isolation level is REPEATABLE READ.
SORT_SEQUENCE	SRTSEQ	VARCHAR(12)	Indicates whether the materialize query table uses a collating sequence: **BY HEX VALUE** The materialize query table does not use a collating table. ***LANGIDSHR** The materialize query table uses a shared weight sort sequence (SRTSEQ). ***LANGIDUNQ** The materialize query table uses a unique weight sort sequence (SRTSEQ). **ALTSEQ** The materialize query table uses an alternate collating sequence (ALTSEQ).
LANGUAGE_IDENTIFIER	LANGID	CHAR(3) Nullable	The language ID of the materialize query table. Contains null if the sort sequence is hex.
SORT_SEQUENCE_SCHEMA	SRTSEQSCH	CHAR(10) Nullable	Schema name of the sort sequence to use. Contains null if there is no schema name.
SORT_SEQUENCE_NAME	SRTSEQNAM	CHAR(10) Nullable	Name of the sort sequence to use. Contains null if there is no sort sequence name.
MQT_RESTORE_DEFERRED	MQTRSTDFR	VARCHAR(3)	An indication of whether a restore of the MQT is pending the restore of one of its dependents: **NO** The restore of the MQT is not deferred pending the restore of one of its dependent tables. **YES** The restore of the MQT is deferred pending the restore of one of its dependent tables.

Table 175. SYSPARTITIONMQTS view (continued)

Column name	System Column Name	Data Type	Description
ROUNDING_MODE	DECFLTRND	VARCHAR(8) Nullable	Indicates the DECFLOAT rounding mode of the materialized query table: **CEILING** ROUND_CEILING **DOWN** ROUND_DOWN **FLOOR** ROUND_FLOOR **HALFDOWN** ROUND_HALF_DOWN **HALFEVEN** ROUND_HALF_EVEN **HALFUP** ROUND_HALF_UP **UP** ROUND_UP Contains the null value if the materialized query table does not have an expression that references a DECFLOAT column, function, or constant.
DECFLOAT_WARNING	DECFLTWRN	VARCHAR(3) Nullable	Indicates whether DECFLOAT warnings are returned: **NO** DECFLOAT warnings are not returned. **YES** DECFLOAT warnings are returned. Contains the null value if the materialized query table does not have an expression that references a DECFLOAT column, function, or constant.
MQT_DEFINITION	MQTDEF	VARGRAPHIC(5000) CCSID 1200	The query of the materialized query table. If the length of the query exceeds 5000, '...' is returned at the end of the column value.
MQT_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	Text of the materialized query table. Contains null if text does not exist for the materialized query table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_MEMBER	SYS_MNAME	CHAR(10)	System member name.

SYSPARTITIONSTAT

The SYSPARTITIONSTAT view contains one row for every table partition or table member. If the table is a distributed table, the partitions that reside on other database nodes are not contained in this catalog view. They are contained in the catalog views of the other database nodes.

The following table describes the columns in the SYSPARTITIONSTAT view:

Table 176. SYSPARTITIONSTAT view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
TABLE_PARTITION	TABPART	VARCHAR(128)	Name of the table partition or member.
PARTITION_TYPE	PARTTYPE	CHAR(1)	The type of the table partitioning: **blank** The table is not partitioned. **H** This is data hash partitioning. **R** This is data range partitioning. **D** This is distributed database hash partitioning.
PARTITION_NUMBER	PARTNBR	INTEGER Nullable	The partition number of this partition. If the table is a distributed table, contains null.
NUMBER_DISTRIBUTED_PARTITIONS	DSTPARTS	INTEGER Nullable	If the table is a distributed table, contains the total number of partitions. If the table is not a distributed table, contains null.
NUMBER_ROWS	CARD	BIGINT	Number of valid rows in the table partition or member.
NUMBER_ROW_PAGES	NPAGES	BIGINT	Number of 64K pages in the partition's data.
NUMBER_PAGES	FPAGES	BIGINT	Same as NUMBER_ROW_PAGES.
OVERFLOW	OVERFLOW	BIGINT	The estimated number of rows that have overflowed to variable length segments. If the table does not contain variable length or LOB columns, contains 0.
CLUSTERED	CLUSTERED	CHAR(1) Nullable	Not applicable for DB2 for i. Will always be null.
ACTIVE_BLOCKS	ACTBLOCKS	BIGINT	Not applicable for DB2 for i. Will always be -1.
AVGCOMPRESSEDROWSIZE	ACROWSIZE	BIGINT	Not applicable for DB2 for i. Will always be -1.
AVGROWCOMPRESSIONRATIO	ACROWRATIO	REAL	Not applicable for DB2 for i. Will always be -1.
AVGROWSIZE	AVGROWSIZE	BIGINT	Average length (in bytes) of a row in this table. If the table has variable length or LOB columns, contains -1.
PCTROWSCOMPRESSED	PCTCROWS	REAL	Not applicable for DB2 for i. Will always be -1.
PCTPAGESSAVED	PCTPGSAVED	SMALLINT	Not applicable for DB2 for i. Will always be -1.
NUMBER_DELETED_ROWS	DELETED	BIGINT	Number of deleted rows in the table partition or member.
DATA_SIZE	SIZE	BIGINT	Total size (in bytes) of the data space in the partition or member.
VARIABLE_LENGTH_SIZE	VLSIZE	BIGINT	Size (in bytes) of the variable-length data space segments in the partition or member.
FIXED_LENGTH_EXTENTS	FLEXTENTS	BIGINT	Number of fixed-length data space segment extents in the partition or member.
VARIABLE_LENGTH_EXTENTS	VLEXTENTS	BIGINT	Number of variable-length data space segment extents in the partition or member.
COLUMN_STATS_SIZE	CSTATSSIZE	BIGINT	Size (in bytes) of the column statistics in the partition or member.

Table 176. SYSPARTITIONSTAT view (continued)

Column name	System Column Name	Data Type	Description
MAINTAINED_TEMPORARY_INDEX_SIZE	MTISIZE	BIGINT	Size (in bytes) of all maintained temporary indexes over the partition or member.
NUMBER_DISTINCT_INDEXES	DISTINCTIX	INTEGER	The number of distinct indexes built over the partition or member. This does not include maintained temporary indexes.
OPEN_OPERATIONS	OPENS	BIGINT	Number of full opens of the partition or member since the last IPL.
CLOSE_OPERATIONS	CLOSES	BIGINT	Number of full closes of the partition or member since the last IPL.
INSERT_OPERATIONS	INSERTS	BIGINT	Number of inserts operations for the partition or member since the last IPL.
UPDATE_OPERATIONS	UPDATES	BIGINT	Number of update operations for the partition or member since the last IPL.
DELETE_OPERATIONS	DELETES	BIGINT	Number of delete operations for the partition or member since the last IPL.
CLEAR_OPERATIONS	DSCLEARS	BIGINT	Number of clear operations (CLRPFM operations) for the partition or member since the last IPL.
COPY_OPERATIONS	DSCOPIES	BIGINT	Number of data space copy operations (certain CPYxxx operations) for the partition or member since the last IPL.
REORGANIZE_OPERATIONS	DSREORGS	BIGINT	Number of data space reorganize operations (non-interruptible RGZPFM operations) for the partition or member since the last IPL.
INDEX_BUILDS	DSINXBLDS	BIGINT	Number of creates or rebuilds of indexes that reference the partition or member since the last IPL. This does not include maintained temporary indexes.
LOGICAL_READS	LGLREADS	BIGINT	Number of logical read operations for the partition or member since the last IPL.
PHYSICAL_READS	PHYREADS	BIGINT	Number of physical read operations for the partition or member since the last IPL.
SEQUENTIAL_READS	SEQREADS	BIGINT	Number of sequential read operations for the partition or member since the last IPL.
RANDOM_READS	RANREADS	BIGINT	Number of random read operations for the partition or member since the last IPL.
CREATE_TIMESTAMP	CREATED	TIMESTAMP	Create timestamp of the partition or member.
LAST_CHANGE_TIMESTAMP	LASTCHG	TIMESTAMP	Timestamp of the last change that occurred to the partition or member.
LAST_SAVE_TIMESTAMP	LASTSAVE	TIMESTAMP Nullable	Timestamp of the last save of the partition or member. If the partition or member has never been saved, contains null.
LAST_RESTORE_TIMESTAMP	LASTRST	TIMESTAMP Nullable	Timestamp of the last restore of the partition or member. If the partition or member has never been restored, contains null.
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP Nullable	Timestamp of the last time the partition or member was used directly by an application for native record I/O or SQL operations. If the partition or member has never been used, contains null.
DAYS_USED_COUNT	DAYSUSED	INTEGER	The number of days the partition or member was used directly by an application for native record I/O or SQL operations since the last time the usage statistics were reset. If the partition or member has never been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	The timestamp of the last time the usage statistics were reset for the table. For more information see the Change Object Description (CHGOBJD) command. If the partition or member's last used timestamp has never been reset, contains null.

Table 176. SYSPARTITIONSTAT view (continued)

Column name	System Column Name	Data Type	Description
NEXT_IDENTITY_VALUE	NEXTVALUE	DECIMAL(31,0) Nullable	The next identity value. In some cases, this value may be an estimate. If the table does not have an identity value, contains null.
LOWINCLUSIVE	LOWINCL	CHAR(1) Nullable	Indicates whether the low key value for the partition is inclusive. **N** The low key value is not inclusive. **Y** The low key value is inclusive. If the table is not partitioned by range, contains null.
LOWVALUE	LOWVALUE	VARGRAPHIC(1024) CCSID 1200 Nullable	A string representation of the low key value for a range partition. If the table is not partitioned by range, contains null.
HIGHINCLUSIVE	HIGHINCL	CHAR(1) Nullable	Indicates whether the high key value for the partition is inclusive. **N** The high key value is not inclusive. **Y** The high key value is inclusive. If the table is not partitioned by range, contains null.
HIGHVALUE	HIGHVALUE	VARGRAPHIC(1024) CCSID 1200 Nullable	A string representation of the high key value for a range partition. If the table is not partitioned by range, contains null.
NUMBER_PARTITIONING_KEYS	NBRPKEYS	INTEGER Nullable	The number of partitioning keys. If the table is not partitioned, contains null.
PARTITIONING_KEYS	PARTKEYS	VARCHAR(2880) Nullable	The list of partitioning keys. If the table is not partitioned, contains null.
KEEP_IN_MEMORY	KEEPINMEM	CHAR(1)	Indicates whether the index should be kept in memory: **0** No memory preference. **1** The index should be kept in memory, if possible.
MEDIA_PREFERENCE	MEDIAPREF	SMALLINT	Indicates the media preference of the index: **0** No media preference. **255** The index should be allocated on Solid State Disk (SSD), if possible.
LAST_SOURCE_UPDATE_TIMESTAMP	LASRSRCUPD	TIMESTAMP Nullable	Last source change timestamp to a source member. If the table is not a source file, contains null.
SOURCE_TYPE	SRCTYPE	VARCHAR(10) Nullable	Source type of a source member. If the table is not a source file, contains null.
VOLATILE	VOLATILE	CHAR(1) Nullable	Indicates whether the table is volatile. If the table is not an SQL table or physical file, contains null.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_MEMBER	SYS_MNAME	CHAR(10)	System member name.

SYSPROCS

The SYSPROCS view contains one row for each procedure created by the CREATE PROCEDURE statement.

The following table describes the columns in the SYSPROCS view:

Table 177. SYSPROCS view

Column Name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSCHEMA	VARCHAR(128)	Schema name of the routine (procedure) instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
ROUTINE_SCHEMA	PROCSCHEMA	VARCHAR(128)	Name of the SQL schema (schema) that contains the routine.
ROUTINE_NAME	PROCNAME	VARCHAR(128)	Name of the routine.
ROUTINE_CREATED	RTNCREATE	TIMESTAMP	Identifies the timestamp when the routine was created.
ROUTINE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the routine.
ROUTINE_BODY	BODY	VARCHAR(8)	The type of the routine body: **EXTERNAL** This is an external routine. **SQL** This is an SQL routine.
EXTERNAL_NAME	EXTNAME	VARCHAR(279) Nullable	This column identifies the external program name. • For ILE service programs, the external program name is *schema-name/service-program-name(entry-point-name)*. • For REXX, the external program name is *schema-name/source-file-name(member-name)*. • For Java programs, the external program name is an optional jar-id followed by a *fully-qualified-class-name!method-name* or *fully-qualified-class-name.method-name*. • For all other languages, the external program name is *schema-name/program-name*.

Table 177. SYSPROCS view (continued)

Column Name	System Column Name	Data Type	Description
EXTERNAL_LANGUAGE	LANGUAGE	VARCHAR(8)	If this is an external routine, this column identifies the external program name.
		Nullable	**C** The external program is written in C.
			C++ The external program is written in C++.
			CL The external program is written in CL.
			COBOL The external program is written in COBOL.
			COBOLLE The external program is written in ILE COBOL.
			FORTRAN The external program is written in FORTRAN.
			JAVA The external program is written in JAVA.
			PLI The external program is written in PL/I.
			REXX The external program is a REXX procedure.
			RPG The external program is written in RPG.
			RPGLE The external program is written in ILE RPG.
			Contains the null value if this is not an external routine.
PARAMETER_STYLE	PARM_STYLE	VARCHAR(7)	If this is an external routine, this column identifies the parameter style (calling convention).
		Nullable	**DB2GNRL** This is the DB2GENERAL calling convention.
			DB2SQL This is the DB2SQL calling convention.
			GENERAL This is the GENERAL calling convention.
			JAVA This is the JAVA calling convention.
			NULLS This is the GENERAL WITH NULLS calling convention.
			SQL This is the SQL standard calling convention.
			Contains the null value if this is not an external routine.
IS_DETERMINISTIC	DETERMINE	VARCHAR(3)	This column identifies whether the routine is deterministic. That is, whether a call to the routine with the same arguments will always return the same result.
			NO The routine is not deterministic.
			YES The routine is deterministic.

Table 177. SYSPROCS view (continued)

Column Name	System Column Name	Data Type	Description
SQL_DATA_ACCESS	DATAACCESS	VARCHAR(8)	This column identifies whether a routine contains SQL and whether it reads or modifies data. **NONE** The routine does not contain any SQL statements. **CONTAINS** The routine contains SQL statements. **READS** The routine possibly reads data from a table or view. **MODIFIES** The routine possibly modifies data in a table or view or issues SQL DDL statements.
SQL_PATH	SQL_PATH	VARCHAR(3483) Nullable	If this is an SQL routine, this column identifies the path. Contains the null value if this is not an SQL routine.
PARM_SIGNATURE	SIGNATURE	VARCHAR(2048)	This column identifies the routine signature.
RESULT_SETS	RESULTS	SMALLINT	Identifies the maximum number of result sets returned. 0 indicates that there are no result sets.
IN_PARMS	IN_PARMS	SMALLINT	Identifies the number of input parameters. 0 indicates that there are no input parameters.
OUT_PARMS	OUT_PARMS	SMALLINT	Identifies the number of output parameters. 0 indicates that there are no output parameters.
INOUT_PARMS	INOUT_PARM	SMALLINT	Identifies the number of input/output parameters. 0 indicates that there are no input/output parameters.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
ROUTINE_DEFINITION	ROUTINEDEF	DBCLOB(2M) CCSID 13488 Nullable	If this is an SQL routine, this column contains the SQL routine body. If this is an obfuscated routine, the text starts with the WRAPPED keyword and is followed by the encoded form of the statement text. Contains the null value if this is not an SQL routine.
DBINFO	DBINFO	VARCHAR(3) Nullable	Identifies whether information about the database is passed to the procedure. **NO** No database information is passed to the procedure. **YES** Information about the database is passed to the procedure.
COMMIT_ON_RETURN	CMTONRET	VARCHAR(3) Nullable	This column identifies whether the procedure commits on a successful return from the procedure. **NO** A commit is not performed on successful return from the procedure. **YES** A commit is performed on successful return from the procedure.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
NEW_SAVEPOINT_LEVEL	NEWSAVEPTL	VARCHAR(3) Nullable	This column identifies whether the routine starts a new savepoint level. **NO** A new savepoint level is not started. **YES** A new savepoint level is started.

Table 177. SYSPROCS view (continued)

Column Name	System Column Name	Data Type	Description
ROUNDING_MODE	DECFLTRND	CHAR(1) Nullable	If this is an SQL procedure, identifies the DECFLOAT rounding mode. **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP Contains the null value if the procedure is not an SQL procedure.
ROUTINE_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	Contains the label for a routine. Contains the null value if a label does not exist.

SYSPROGRAMSTAT

The SYSPROGRAMSTAT view contains one row for each program, service program, and module that contains SQL statements.

The following table describes the columns in the SYSPROGRAMSTAT view:

Table 178. SYSPROGRAMSTAT view

Column Name	System Column Name	Data Type	Description
PROGRAM_SCHEMA	COLLID	VARCHAR(128)	Name of the schema
PROGRAM_NAME	NAME	VARCHAR(128)	Name of the program, service program, or module
PROGRAM_TYPE	PGMTYPE	VARCHAR(128)	Type of the object ***PGM** The object is a program. ***MODULE** The object is a module. ***SRVPGM** The object is a service program.
PROGRAM_OWNER	OWNER	VARCHAR(128)	Owner of the program, service program, or module
PROGRAM_CREATOR	CREATOR	VARCHAR(128)	Creator of the program, service program, or module
CREATION_TIMESTAMP	TIMESTAMP	TIMESTAMP	Timestamp of when the program, service program, or module was created
DEFAULT_SCHEMA	QUALIFIER	VARCHAR(128) Nullable	Implicit name for unqualified tables, views, and indexes. Contains null if a default schema was not specified (DFTRDBCOL) or if the program is an external procedure without SQL statements.
ISOLATION	ISOLATION	CHAR(2) Nullable	Isolation option specification: **RR** Repeatable Read (*RR) **RS** Read Stability (*ALL) **CS** Cursor Stability (*CS) **UR** Uncommitted Read (*CHG) **NC** No Commit (*NONE) Contains null if the program is an external procedure without SQL statements.
CONCURRENTACCESSRESOLUTION	CONCURRENT	CHAR(1) Nullable	Specifies the concurrent access resolution: **blank** Not specified **W** Wait for outcome **U** Use currently committed Contains null if the program is an external procedure without SQL statements.
NUMBER_STATEMENTS	NBRSTMTS	INTEGER	Number of SQL statements in the program, service program or module
PROGRAM_USED_SIZE	PGMSIZE	INTEGER	Number of bytes that are used for SQL statements and access plans in the program, service program or module.
NUMBER_COMPRESSIONS	PGM_CMP Nullable	INTEGER Nullable	Number of times the program or service program has been compressed. Contains null for modules or if the program is an external procedure without SQL statements.
STATEMENT_CONTENTION_COUNT	CONTENTION	BIGINT Nullable	Number of times contention occurred when attempting to store a new access plan. Contains null for modules or if the program is an external procedure without SQL statements.

Table 178. SYSPROGRAMSTAT view (continued)

Column Name	System Column Name	Data Type	Description
ORIGINAL_SOURCE_FILE	SOURCE	VARCHAR(128) Nullable	The fully qualified source file and member that was used to create the program or module. Contains null for SQL routines or if the program is an external procedure without SQL statements.
ORIGINAL_SOURCE_FILE_CCSID	SRC_CCSID	INTEGER Nullable	The CCSID of the source file that was used to create the program or module. Contains null for SQL routines or if the program is an external procedure without SQL statements.
ROUTINE_TYPE	RTNTYPE	VARCHAR(9) Nullable	Type of the routine. **PROCEDURE** This is a procedure. **FUNCTION** This is a function. **TRIGGER** This is a trigger. Contains null for modules or if the program or service program is not a procedure, function, or trigger. An external procedure will not be identified as PROCEDURE unless NUMBER_EXTERNAL_ROUTINES is greater than zero.
ROUTINE_BODY	BODY	VARCHAR(8) Nullable	The type of the routine body: **EXTERNAL** This is an external routine. **SQL** This is an SQL routine. Contains null for modules or if the program or service program is not a procedure or function.
FUNCTION_ORIGIN	ORIGIN	CHAR(1) Nullable	Identifies the type of function. If this is a procedure, this column contains a blank. **B** This is a built-in function (defined by DB2 for i). **E** This is a user-defined function. **U** This is a user-defined function that is sourced on another function. **S** This is a system-generated function. Contains null for modules or if the program or service program is not a procedure or function.
FUNCTION_TYPE	TYPE	CHAR(1) Nullable	Identifies the form of the function. If this is a procedure, this column contains a blank. **S** This is a scalar function. **C** This is a column function. **T** This is a table function. Contains null for modules or if the program or service program is not a procedure, function, or trigger.
NUMBER_EXTERNAL_ROUTINES	NBREXTRTN	SMALLINT Nullable	Indicates the number of procedure and function definitions stored in the program or service program. Contains null for modules, triggers, or SQL routines.

Table 178. SYSPROGRAMSTAT view (continued)

Column Name	System Column Name	Data Type	Description
EXTENDED_INDICATOR	EXTIND	VARCHAR(9) Nullable	Indicates the EXTIND attribute: ***EXTIND** Extended indicator support is enabled. ***NOEXTIND** Extended indicator support is not enabled. Contains null if the program is an external procedure without SQL statements.
C_NUL_REQUIRED	CNULRQD	VARCHAR(10) Nullable	Indicates the CNULRQD attribute: ***CNULRQD** C nuls are required. ***NOCNULRQD** C nuls are not required. Contains null if the program is an external procedure without SQL statements.
NAMING	NAMING	VARCHAR(4) Nullable	Indicates the NAMING attribute: ***SYS** This is system naming. ***SQL** This is SQL naming. Contains null if the program is an external procedure without SQL statements.
TARGET_RELEASE	TGTRLS	VARCHAR(6) Nullable	Indicates the target release of the program, service program, or module (VxRxMx). Contains null if the program is an external procedure without SQL statements.
EARLIEST_POSSIBLE_RELEASE	MINRLS	VARCHAR(6) Nullable	Indicates the earliest IBM i release that supports all the SQL statements in the program, service program, or module (VxRxMx). **ANY** The statements are valid on any supported IBM i release. **VxRxMx** The statement is valid on IBM i VxRxMx release or later. Contains null if the earliest release has not yet been determined or if the program is an external procedure without SQL statements.
RDB	RDB	VARCHAR(18) Nullable	Indicates the RDB specified for the program, service program, or module. **rdb-name** The name of the relational database. ***NONE** A relational database was not specified. Contains null if the program is an external procedure without SQL statements.
ALLOW_COPY_DATA	ALWCPYDTA	VARCHAR(9) Nullable	Indicates the ALWCPYDTA attribute: ***NO** A copy of the data is not allowed. ***OPTIMIZE** A copy of the data is allowed whenever it might result in better performance. ***YES** A copy of the data is allowed, but only when necessary. Contains null if the program is an external procedure without SQL statements.

Table 178. SYSPROGRAMSTAT view (continued)

Column Name	System Column Name	Data Type	Description
CLOSE_SQL_CURSOR	CLOSQLCSR	VARCHAR(10) Nullable	Indicates the CLOSQLCSR attribute: ***ENDACTGRP** SQL cursors are closed and SQL prepared statements are implicitly discarded, and LOCK TABLE locks are released when the activation group ends. ***ENDJOB** SQL cursors are closed and SQL prepared statements are implicitly discarded, and LOCK TABLE locks are released when the job ends. ***ENDMOD** SQL cursors are closed and SQL prepared statements are implicitly discarded when the module is exited. LOCK TABLE locks are released when the first SQL program on the call stack ends. ***ENDPGM** SQL cursors are closed and SQL prepared statements are implicitly discarded when the program ends. LOCK TABLE locks are released when the first SQL program on the call stack ends. Contains null if the program is an external procedure without SQL statements.
LOB_FETCH_OPTIMIZATION	OPTLOB	VARCHAR(9)	Indicates the LOB optimization attribute: ***OPTLOB** The first FETCH for a cursor determines how the cursor will be used for LOB and XML result columns on all subsequent FETCHes. ***NOOPTLOB** Any FETCH may retrieve a LOB or XML result column into either a locator or variable.
DECIMAL_POINT	DECPNT	VARCHAR(7)	Indicates the decimal point for numeric constants used in SQL statements. ***PERIOD** The decimal point is a period. ***COMMA** The decimal point is a comma.
SQL_STRING_DELIMITER	STRDLM	VARCHAR(9)	Indicates the character used as the string delimiter in the SQL statements. ***APOSTSQL** The string delimiter is an apostrophe ('). ***QUOTESQL** The string delimiter is a quote (").

Table 178. SYSPROGRAMSTAT view (continued)

Column Name	System Column Name	Data Type	Description
DATE_FORMAT	DATFMT	VARCHAR(4) Nullable	Indicates the DATFMT attribute: ***JOB** The date format specified in the job at runtime is used. ***USA** The date format is *USA. ***ISO** The date format is *ISO. ***EUR** The date format is *EUR. ***JIS** The date format is *JIS. ***MDY** The date format is *MDY. ***DMY** The date format is *DMY. ***YMD** The date format is *YMD. ***JUL** The date format is *JUL. Contains null if the program is an external procedure without SQL statements.
DATE_SEPARATOR	DATSEP	CHAR(1) Nullable	Indicates the date separator. Contains null if the program is an external procedure without SQL statements.
TIME_FORMAT	TIMFMT	VARCHAR(4) Nullable	Indicates the TIMFMT attribute: ***JOB** The time format specified in the job at runtime is used. ***USA** The time format is *USA. ***ISO** The time format is *ISO. ***EUR** The time format is *EUR. ***JIS** The time format is *JIS. ***HMS** The date format is *HMS. Contains null if the program is an external procedure without SQL statements.
TIME_SEPARATOR	TIMSEP	CHAR(1) Nullable	Indicates the time separator. Contains null if the program is an external procedure without SQL statements.
DYNAMIC_DEFAULT_SCHEMA	DYNDFTCOL	VARCHAR(4) Nullable	Indicates whether the value for DFTRDBCOL should be used for implicit qualification on dynamic SQL statements: ***NO** The schema specified in DFTDRBCOL is not used for dynamic SQL statements. ***YES** The schema specified in DFTDRBCOL is used for dynamic SQL statements. Contains null if a default schema was not specified (DFTRDBCOL) or if the program is an external procedure without SQL statements.
CURRENT_RULES	SQLCURRULE	VARCHAR(4) Nullable	Indicates the SQLCURRULE attribute: ***DB2** The semantics of all SQL statements will default to the rules established for DB2. ***STD** The semantics of all SQL statements will default to the rules established by the ISO and ANSI SQL standards. Contains null if the program is an external procedure without SQL statements.

Table 178. SYSPROGRAMSTAT view (continued)

Column Name	System Column Name	Data Type	Description
ALLOW_BLOCK	ALWBLK	VARCHAR(8) Nullable	Indicates the ALWBLK attribute: ***ALLREAD** Rows are blocked for read-only cursors. ***NONE** Rows are not blocked for retrieval of data for cursors. ***READ** Records are blocked for read-only retrieval of data for cursors when: • *NONE is specified for the Commitment control (COMMIT) parameter. • The cursor is declared with a FOR READ ONLY clause or there are no dynamic statements that could run a positioned UPDATE or DELETE statement for the cursor. Contains null if the program is an external procedure without SQL statements.
DELAY_PREPARE	DLYPRP	VARCHAR(4) Nullable	Indicates the DLYPRP attribute: ***NO** Dynamic statement validation is performed when the dynamic statements are prepared. ***YES** Dynamic statement validation is delayed until the dynamic statements are used. Contains null if the program is an external procedure without SQL statements.
USER_PROFILE	USRPRF	VARCHAR(7) Nullable	Specifies the user profile used for authority checking: ***USER** The profile of the user running the program is used. ***OWNER** The profiles of both the owner of the program and the user running the program is used. ***NAMING** If the naming convention is *SQL, *OWNER is used. If the naming convention is *SYS, *USER is used. Contains null if the program is an external procedure without SQL statements.
DYNAMIC_USER_PROFILE	DYNUSRPRF	VARCHAR(6) Nullable	Specifies the user profile used for dynamic SQL statements: ***USER** Local dynamic SQL statements are run under the profile of the job or thread. Distributed dynamic SQL statements are run under the profile of the application server job. ***OWNER** Local dynamic SQL statements are run under the profile of the program's owner. Distributed dynamic SQL statements are run under the profile of the SQL package's owner. Contains null if the program is an external procedure without SQL statements.

Table 178. SYSPROGRAMSTAT view (continued)

Column Name	System Column Name	Data Type	Description
SORT_SEQUENCE	SRTSEQ	VARCHAR(12)	Indicates whether the program, service program, or module uses a collating sequence:
		Nullable	**BY HEX VALUE** The SQL index does not use a collating table.
			***LANGIDSHR** The SQL index uses a shared weight sort sequence (SRTSEQ).
			***LANGIDUNQ** The SQL index uses a unique weight sort sequence (SRTSEQ).
			ALTSEQ The SQL index uses an alternate collating sequence (ALTSEQ).
			Contains null if the program is an external procedure without SQL statements.
LANGUAGE_IDENTIFIER	LANGID	CHAR(3)	The language ID sort sequence.
		Nullable	Contains null if the sort sequence is not *LANGIDSHR or *LANGIDUNQ or if the program is an external procedure without SQL statements.
SORT_SEQUENCE_SCHEMA	SRTSEQSCH	CHAR(10)	The sort sequence table system schema.
		Nullable	Contains null if the sort sequence is hex or if the program is an external procedure without SQL statements.
SORT_SEQUENCE_NAME	SRTSEQNAME	CHAR(10)	The sort sequence table name.
		Nullable	Contains null if the sort sequence is hex or if the program is an external procedure without SQL statements.
RDB_CONNECTION_METHOD	RDBCNNMTH	VARCHAR(4)	Specifies the semantics used for CONNECT statements:
		Nullable	***RUW** CONNECT (Type 1) semantics are used to support remote unit of work.
			***DUW** CONNECT (Type 2) semantics are used to support distributed unit of work.
			Contains null if the program is an external procedure without SQL statements.
DECRESULT_MAXIMUM_PRECISION	DECMAXPRC	SMALLINT	Specifies the maximum precision.
		Nullable	**31** The maximum precision is 31.
			63 The maximum precision is 63.
			Contains null if the program is an external procedure without SQL statements.
DECRESULT_MAXIMUM_SCALE	DECMAXSCL	SMALLINT	The maximum scale (number of decimal positions to the right of the decimal point) that should be returned for result data types.
		Nullable	Contains null if the program is an external procedure without SQL statements.
DECRESULT_MINIMUM_DIVIDE_SCALE	DECMINDIV	SMALLINT	The minimum divide scale (number of decimal positions to the right of the decimal point) that should be returned for both intermediate and result data types.
		Nullable	Contains null if the program is an external procedure without SQL statements.

Table 178. SYSPROGRAMSTAT view (continued)

Column Name	System Column Name	Data Type	Description
DECFLOAT_ROUNDING_MODE	DECFLTRND	VARCHAR(8) Nullable	Indicates the DECFLOAT rounding mode: **CEILING** ROUND_CEILING **DOWN** ROUND_DOWN **FLOOR** ROUND_FLOOR **HALFDOWN** ROUND_HALF_DOWN **HALFEVEN** ROUND_HALF_EVEN **HALFUP** ROUND_HALF_UP **UP** ROUND_UP Contains null if the program is an external procedure without SQL statements.
DECFLOAT_WARNING	DECFLTWRN	VARCHAR(3) Nullable	Indicates whether DECFLOAT warnings are returned. **NO** DECFLOAT warnings are not returned. **YES** DECFLOAT warnings are returned. Contains null if the program is an external procedure without SQL statements.
SQLPATH	SQLPATH	VARCHAR(3483) Nullable	Identifies the SQL path. Contains the null value if an SQL path is not specified or if the program is an external procedure without SQL statements.
DBGVIEW	DBGVIEW	VARCHAR(9) Nullable	Specifies the type of source debug information: ***NONE** No debug. ***SOURCE** Debug view includes source and SQL INCLUDE statements. ***STMT** Debug view includes precompiler generated statements. ***LIST** Debug view includes the compiled listing. ***LSTDBG** Debug view includes the compiled listing of an OPM program. **ALLOW** Source debug allowed by the Unified Debugger. **DISALLOW** Source debug not allowed by the Unified Debugger. **DISABLE** Source debug not allowed by the Unified Debugger and the DEBUG MODE cannot be altered. Contains null if the program is an external procedure without SQL statements.

Table 178. SYSPROGRAMSTAT view (continued)

Column Name	System Column Name	Data Type	Description
DBGKEY	DBGKEY	VARCHAR(3) Nullable	Specifies the type of source debug information: **NO** No encryption key was specified on the debug encryption key (DBGENCKEY) parameter. **YES** A key was specified on the debug encryption key (DBGENCKEY) parameter. Contains null if DBGENCKEY is not supported or if the program is an external procedure without SQL statements.
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP Nullable	The timestamp of the last time the program, service program, or module was used. Contains null if the program, service program, or module has never been used.
DAYS_USED_COUNT	DAYSUSED	INTEGER	The number of days the program, service program, or module was used since the last time the usage statistics were reset. If the program, service program, or module has never been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	The timestamp of the last time the usage statistics were reset. Contains null if the statistics have never been reset.
SYSTEM_PROGRAM_NAME	SYS_NAME	CHAR(10)	System name of the program, service program, or module.
SYSTEM_PROGRAM_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the program, service program, or module.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSPROGRAMSTMTSTAT

The SYSPROGRAMSTMTSTAT view contains one row for each embedded SQL statement in a program, module, or service program.

The following table describes the columns in the SYSPROGRAMSTMTSTAT view:

Table 179. SYSPROGRAMSTMTSTAT view

Column Name	System Column Name	Data Type	Description
PROGRAM_SCHEMA	COLLID	VARCHAR(128)	Name of the schema.
PROGRAM_NAME	NAME	VARCHAR(128)	Name of the program.
PROGRAM_TYPE	PGMTYPE	VARCHAR(128)	Type of the object : ***PGM** The object is a program. ***MODULE** The object is a module. ***SRVPGM** The object is a service program.
MODULE_NAME	MODNAME	VARCHAR(10) Nullable	Module name for ILE program or service program. Contains the null value if this is not an ILE program or service program.
STATEMENT_NUMBER	STMTNBR	INTEGER	Statement number in program.
NUMBER_TIMES_EXECUTED	NBREXEC	INTEGER	Number of times this statement has been executed.
ROWS_AFFECTED	ROWCNT	INTEGER	Total rows fetched, updated, inserted, or deleted for all executions of the statement. Contains 0 if the statement is not a FETCH, SET, VALUES INTO, UPDATE, INSERT, DELETE, or MERGE. Will contain 0 for a SET or VALUES INTO statement that is not implemented using a cursor.
NUMBER_HOST_VARIABLES	NBRHV	INTEGER	Total number of host variables specified in the statement. This includes input and output host variables.
NUMBER_INPUT_HOST_VARIABLES	NBRIHV	INTEGER	Number of input host variables specified in the statement.
WITH_HOLD	WITHHOLD	CHAR(3) Nullable	Specifies the WITH HOLD option for statement: **YES** WITH HOLD clause specified. Contains the null value if the clause was not specified.
FETCH_ONLY	FETCHONLY	CHAR(3) Nullable	Specifies the FOR READ ONLY option for statement: **YES** FOR READ ONLY clause specified. Contains the null value if the clause was not specified.
CONCURRENTACCESSRESOLUTION	CONCURRENT	CHAR(1) Nullable	Specifies the concurrent access resolution for the statement: **W** Wait for outcome **U** Use currently committed **S** Skip locked data Contains the null value if concurrent access was not specified at the statement level.
NUMBER_REBUILDS	NBRREBLD	INTEGER	Number of times QDT or access plan has been rebuilt.

Table 179. SYSPROGRAMSTMTSTAT view (continued)

Column Name	System Column Name	Data Type	Description
ISOLATION	ISOLATION	CHAR(2) Nullable	Isolation option specification: **RR** Repeatable Read (*RR) **RS** Read Stability (*ALL) **CS** Cursor Stability (*CS) **UR** Uncommitted Read (*CHG) **NC** No Commit (*NONE) Contains the null value if isolation level was not specified at the statement level.
NUMBER_ROWS_TO_OPTIMIZE	OPTROWS	INTEGER Nullable	Number of rows specified on the OPTIMIZE FOR *n* ROWS clause. -1 means that the value *ALL was specified. Contains the null value if the clause was not specified.
NUMBER_ROWS_TO_FETCH	FETCHROWS	INTEGER Nullable	Number of rows specified on the FETCH FIRST *n* ROWS clause. Contains the null value if the clause was not specified.
LAST_QDT_REBUILD_REASON	QDTRBLD	CHAR(2) Nullable	Reason code for last QDT rebuild. This corresponds to column QVC22 in the STRDBMON outfile when QQRID = 1000. Contains the null value if the statement does not use a QDT or has never had a QDT rebuilt.
STATEMENT_TEXT	STMTTEXT	DBCLOB(2M) CCSID(1200)	Text of the SQL statement.
SYSTEM_PROGRAM_NAME	SYS_NAME	CHAR(10)	System name of the program.
SYSTEM_PROGRAM_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the program.
ACCESS_PLAN_LENGTH	AP_LENGTH	INTEGER	Number of bytes that are used for the QDT and access plan for the statement.
EARLIEST_POSSIBLE_RELEASE	MINRLS	VARCHAR(6)	The earliest IBM i release that supports this SQL statement (VxRxMx). **ANY** The statement is valid on any supported IBM i release. **VxRxMx** The statement is valid on IBM i VxRxMx release or later. Contains the null value if the earliest release is not known.
SQL_DB2_GROUP_LEVEL	SQL_LEVEL	INTEGER Nullable	The latest DB2 for i PTF Group level that SQL language syntax used in this statement is dependent on. Contains the null value if no SQL syntax in this statement was identified as being dependent on a DB2 for i PTF Group.

Table 179. SYSPROGRAMSTMTSTAT view (continued)

Column Name	System Column Name	Data Type	Description
SERVICES_DB2_GROUP_LEVEL	SERV_LEVEL	INTEGER Nullable	The latest DB2 for i PTF Group level that a service referenced in this statement could be dependent on. This column contains a value if any function, table function, view, procedure, or global variable directly referenced in the statement has the same name as one provided by IBM as a service. If the SQL statement includes unqualified references to objects whose names match IBM provided services, this column considers the unqualified name as the IBM service. Built-in functions and built-in global variables that were added or had significant changes in a DB2 for i PTF Group are reflected in this column as well. Contains the null value if no built-in function, built-in global variable, or IBM-supplied service in this statement was identified as possibly being dependent on a DB2 for i PTF Group.

SYSREFCST

The SYSREFCST view contains one row for each foreign key in the SQL schema.

The following table describes the columns in the SYSREFCST view:

Table 180. SYSREFCST view

Column Name	System Column Name	Data Type	Description
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
UNIQUE_CONSTRAINT_SCHEMA	UNQDBNAME	VARCHAR(128) Nullable	Name of the SQL schema containing the unique constraint referenced by the referential constraint. Contains the null value if the unique constraint does not exist. This is usually caused by restoring the table without its parent table.
UNIQUE_CONSTRAINT_NAME	UNQNAME	VARCHAR(128) Nullable	Name of the unique constraint referenced by the referential constraint. Contains the null value if the unique constraint does not exist. This is usually caused by restoring the table without its parent table.
MATCH_OPTION	MATCH	VARCHAR(7)	Match option. Will always be NONE.
UPDATE_RULE	UPDATE	VARCHAR(11)	Update Rule. • NO ACTION • RESTRICT
DELETE_RULE	DELETE	VARCHAR(11)	Delete Rule • NO ACTION • CASCADE • SET NULL • SET DEFAULT • RESTRICT
COLUMN_COUNT	COLCOUNT	INTEGER	Number of columns in the foreign key.
SYSTEM_CONSTRAINT_SCHEMA	SYS_CDNAME	CHAR(10)	Name of the system schema containing the constraint.
SYSTEM_UNIQUE_CONSTRAINT_SCHEMA	SYS_UDNAME	CHAR(10) Nullable	Name of the system schema containing the constraint. Contains the null value if the unique constraint does not exist. This is usually caused by restoring the table without its parent table.

SYSROUTINEAUTH

The SYSROUTINEAUTH view contains one row for every privilege granted on a routine. Note that this catalog view cannot be used to determine whether a user is authorized to a routine because the privilege to use a routine could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the SYSROUTINEAUTH view:

Table 181. SYSROUTINEAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
ROUTINE_SCHEMA	RTNSCHEMA	VARCHAR(128)	Name of the schema
ROUTINE_NAME	RTNNAME	VARCHAR(128)	Name of the routine
SPECIFIC_SCHEMA	SPECSCHEMA	VARCHAR(128)	Specific name of the schema
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the routine. **EXECUTE** The privilege to execute the routine.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	If the privilege is granted through an authorization, contains the name of the authorization list. Contains the null value if the privilege is not granted through an authorization list.

SYSROUTINEDEP

The SYSROUTINEDEP view records the dependencies of routines.

The following table describes the columns in the SYSROUTINEDEP view:

Table 182. SYSROUTINEDEP view

Column name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSCHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
OBJECT_CATALOG	BCATALOG	VARCHAR(128) Nullable Contains the null value if the relational database name was not specified.	Name of the relational database that contains the object.
OBJECT_SCHEMA	BSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the object.
OBJECT_NAME	BNAME	VARCHAR(128)	Name of the object the routine is dependent on.
OBJECT_TYPE	BTYPE	CHAR(24)	Indicates the object type of the object referenced in the routine: **ALIAS** The object is an alias. **FUNCTION** The object is a function. **INDEX** The object is an index. **MATERIALIZED QUERY TABLE** The object is a materialized query table. **PROCEDURE** The object is a procedure. **SCHEMA** The object is a schema. **SEQUENCE** The object is a sequence. **TABLE** The object is a table. If the object does not exist at the time the routine is created or the OBJECT_SCHEMA is *LIBL, TABLE may be returned even though the actual object used at run time may be an alias, materialized query table, or view. **TYPE** The object is a distinct type. **VARIABLE** The object is a variable. **VIEW** The object is a view.
PARM_SIGNATURE	SIGNATURE	VARCHAR(10000) Nullable	This column identifies the routine signature. Contains the null value if the object is not a routine.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number of the object.
NUMBER_OF_PARMS	NUMPARMS	SMALLINT Nullable	Identifies the number of parameters. Contains the null value if the object is not a routine.

SYSROUTINES

The SYSROUTINES table contains one row for each procedure created by the CREATE PROCEDURE statement and each function created by the CREATE FUNCTION statement.

The following table describes the columns in the SYSROUTINES table:

Table 183. SYSROUTINES table

Column Name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSCHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
ROUTINE_SCHEMA	RTNSCHEMA	VARCHAR(128)	Name of the SQL schema (schema) that contains the routine.
ROUTINE_NAME	RTNNAME	VARCHAR(128)	Name of the routine.
ROUTINE_TYPE	RTNTYPE	VARCHAR(9)	Type of the routine. **PROCEDURE** This is a procedure. **FUNCTION** This is a function.
ROUTINE_CREATED	RTNCREATE	TIMESTAMP	Identifies the timestamp when the routine was created.
ROUTINE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the routine.
ROUTINE_BODY	BODY	VARCHAR(8)	The type of the routine body: **EXTERNAL** This is an external routine. **SQL** This is an SQL routine.
EXTERNAL_NAME	EXTNAME	VARCHAR(279) Nullable	This column identifies the external program name. • For SQL functions or ILE service programs, the external program name is *schema-name/service-program-name(entry-point-name)*. • For REXX, the external program name is *schema-name/source-file-name(member-name)*. • For Java programs, the external program name is an optional jar-id followed by a *fully-qualified-class-name!method-name* or *fully-qualified-class-name.method-name*. • For all other languages, the external program name is *schema-name/program-name*. Contains the null value if this is a system-generated function.

Table 183. SYSROUTINES table (continued)

Column Name	System Column Name	Data Type	Description
EXTERNAL_LANGUAGE	LANGUAGE	VARCHAR(8) Nullable	If this is an external routine, this column identifies the external program name. **C** The external program is written in C. **C++** The external program is written in C++. **CL** The external program is written in CL. **COBOL** The external program is written in COBOL. **COBOLLE** The external program is written in ILE COBOL. **FORTRAN** The external program is written in FORTRAN. **JAVA** The external program is written in JAVA. **PLI** The external program is written in PL/I. **REXX** The external program is a REXX procedure. **RPG** The external program is written in RPG. **RPGLE** The external program is written in ILE RPG. Contains the null value if this is not an external routine.
PARAMETER_STYLE	PARM_STYLE	VARCHAR(7) Nullable	If this is an external routine, this column identifies the parameter style (calling convention). **DB2GNRL** This is the DB2GENERAL calling convention. **DB2SQL** This is the DB2SQL calling convention. **GENERAL** This is the GENERAL calling convention. **JAVA** This is the JAVA calling convention. **NULLS** This is the GENERAL WITH NULLS calling convention. **SQL** This is the SQL standard calling convention. Contains the null value if this is not an external routine.
IS_DETERMINISTIC	DETERMINE	VARCHAR(3)	This column identifies whether the routine is deterministic. That is, whether a call to the routine with the same arguments will always return the same result. **NO** The routine is not deterministic. **YES** The routine is deterministic.

Table 183. SYSROUTINES table (continued)

Column Name	System Column Name	Data Type	Description
SQL_DATA_ACCESS	DATAACCESS	VARCHAR(8) Nullable	This column identifies whether a routine contains SQL and whether it reads or modifies data. **NONE** The routine does not contain any SQL statements. **CONTAINS** The routine contains SQL statements. **READS** The routine possibly reads data from a table or view. **MODIFIES** The routine possibly modifies data in a table or view or issues SQL DDL statements.
SQL_PATH	SQL_PATH	VARCHAR(3483) Nullable	If this is an SQL routine, this column identifies the path. Contains the null value if this is not an SQL routine.
PARM_SIGNATURE	SIGNATURE	VARCHAR(2048)	This column identifies the routine signature.
NUMBER_OF_RESULTS	NUMRESULTS	SMALLINT	Identifies the number of results.
MAX_DYNAMIC_RESULT_SETS	RESULTS	SMALLINT	Identifies the maximum number of result sets returned. 0 indicates that there are no result sets.
IN_PARMS	IN_PARMS	SMALLINT	Identifies the number of input parameters. 0 indicates that there are no input parameters.
OUT_PARMS	OUT_PARMS	SMALLINT	Identifies the number of output parameters. 0 indicates that there are no output parameters.
INOUT_PARMS	INOUT_PARM	SMALLINT	Identifies the number of input/output parameters. 0 indicates that there are no input/output parameters.
PARSE_TREE	PARSE_TREE	VARCHAR(1024) FOR BIT DATA	If this is a routine, this column identifies the parse tree of the CREATE FUNCTION or CREATE PROCEDURE statement. It is only used internally.
PARM_ARRAY	PARM_ARRAY	BLOB(320000)	If this is an external routine, this column identifies the parameter array built from the CREATE FUNCTION or CREATE PROCEDURE statement. It is only used internally.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
ROUTINE_DEFINITION	ROUTINEDEF	DBCLOB(2M) CCSID 13488 Nullable	If this is an SQL routine, this column contains the SQL routine body. If this is an obfuscated routine, the text starts with the WRAPPED keyword and is followed by the encoded form of the statement text. Contains the null value if this is not an SQL routine.
FUNCTION_ORIGIN	ORIGIN	CHAR(1)	Identifies the type of function. If this is a procedure, this column contains a blank. **B** This is a built-in function (defined by DB2 for i). **E** This is a user-defined function. **U** This is a user-defined function that is sourced on another function. **S** This is a system-generated function.

Table 183. SYSROUTINES table (continued)

Column Name	System Column Name	Data Type	Description
FUNCTION_TYPE	TYPE	CHAR(1)	Identifies the form of the function. If this is a procedure, this column contains a blank. **S** This is a scalar function. **C** This is a column function. **T** This is a table function.
EXTERNAL_ACTION	EXTACTION	CHAR(1) Nullable	Identifies whether the invocation of the function has external effects. **E** This function has external side effects. **N** This function does not have any external side effects. Contains the null value if the routine is a procedure.
IS_NULL_CALL	NULL_CALL	VARCHAR(3) Nullable	Identifies whether the function needs to be called if an input parameter is the null value. **NO** This function need not be called if an input parameter is the null value. If this is a scalar function, the result of the function is implicitly null if any of the operands are null. If this is a table function, the result of the function is an empty table if any of the operands are the null value. **YES** This function must be called even if an input operand is null. Contains the null value if the routine is a procedure.
SCRATCH_PAD	SCRATCHPAD	INTEGER Nullable	Identifies whether the address of a static memory area (scratch pad) is passed to the function. **0** The function does not have a scratch pad. **integer** Indicates the size of the scratch pad passed to the function. Contains the null value if the routine is a procedure.
FINAL_CALL	FINAL_CALL	VARCHAR(3) Nullable	Indicates whether a final call to the function should be made to allow the function to clean up its work areas (scratch pads). **NO** No final call is made. **YES** A final call to the function is made when the statement is complete. Contains the null value if the routine is a procedure.
PARALLELIZABLE	PARALLEL	VARCHAR(3) Nullable	Identifies whether the function can be run in parallel. **NO** The function must be synchronous. **YES** The function can be run in parallel. Contains the null value if the routine is a procedure.

Table 183. SYSROUTINES table (continued)

Column Name	System Column Name	Data Type	Description
DBINFO	DBINFO	VARCHAR(3) Nullable	Identifies whether information about the database is passed to the routine. **NO** No database information is passed to the routine. **YES** Information about the database is passed to the routine. Contains the null value if the routine is a procedure.
SOURCE_SPECIFIC_SCHEMA	SRCSCHEMA	VARCHAR(128) Nullable	If this is sourced function and the source is user-defined, this column contains the name of the source schema. If this is a sourced function and the source is built-in, this column contains 'QSYS2'. Contains the null value if the routine is not a sourced function.
SOURCE_SPECIFIC_NAME	SRCNAME	VARCHAR(128) Nullable	If this is sourced function and the source is user-defined, this column contains the specific name of the source function name. Contains the null value if the routine is not a sourced function.
IS_USER_ DEFINED_CAST	CAST_FUNC	VARCHAR(3) Nullable	Identifies whether the this function is a cast function created when a distinct type was created. **NO** This function is not a cast function. **YES** This function is a cast function. Contains the null value if the routine is a procedure.
CARDINALITY	CARD	BIGINT Nullable	Specifies the cardinality for a table function. Contains the null value if the function is not a table function or if cardinality was not specified.
FENCED	FENCED	VARCHAR(3) Nullable	Identifies whether a function is fenced. **NO** The function is not fenced. **YES** The function is fenced. Contains the null value if the routine is a procedure.
COMMIT_ON_RETURN	CMTONRET	VARCHAR(3) Nullable	This column identifies whether the procedure commits on a successful return from the procedure. **NO** A commit is not performed on successful return from the procedure. **YES** A commit is performed on successful return from the procedure. Contains the null value if the routine is a function.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
NEW_SAVEPOINT_LEVEL	NEWSAVEPTL	VARCHAR(3) Nullable	This column identifies whether the routine starts a new savepoint level. **NO** A new savepoint level is not started. **YES** A new savepoint level is started. Contains the null value if the routine is a function.
LAST_ALTERED	ALTEREDTS	TIMESTAMP Nullable	Timestamp when routine was last altered. Contains null if the routine has never been altered.

Table 183. SYSROUTINES table (continued)

Column Name	System Column Name	Data Type	Description
DEBUG_MODE	DEBUG_MODE	CHAR(1)	Identifies whether the routine is debuggable. **0** The routine is not debuggable. **1** The routine is debuggable by the Unified Debugger. **2** The routine is debuggable by the system debugger. **N** The routine is disabled from being debugged by the Unified Debugger.
DEBUG_DATA	DEBUG_DATA	CLOB(1048576) Nullable	Reserved. Contains the null value.
ROUNDING_MODE	DECFLTRND	CHAR(1) Nullable	If this is an SQL routine, identifies the DECFLOAT rounding mode. **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP Contains the null value if the routine is not an SQL routine.
ROUTINE_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	Contains the label for a routine. Contains the null value if a label does not exist.
ROUTINE_ENVIRONMENT	RTN_ENV	BLOB(16M) Nullable	Contains internal environment information for a routine defined with default expressions. Contains the null value if this is a procedure or function with no default expressions.
ROUTINE_DEFAULT_QDT	RTNDFTQDT	BLOB(1M) Nullable	Contains internal structure for a routine defined with default expressions. Contains the null value if this is a procedure or function with no default expressions.

SYSSCHEMAAUTH

The SYSSCHEMAAUTH view contains one row for every privilege granted on a schema. Note that this catalog view cannot be used to determine whether a user is authorized to a schema because the privilege to use a schema could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the SYSSCHEMAAUTH view:

Table 184. SYSSCHEMAAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
SCHEMA_NAME	NAME	VARCHAR(128)	Name of the schema
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **CREATEIN** The privilege to create objects in the schema. **USAGE** The privilege to use the schema.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	If the privilege is granted through an authorization, contains the name of the authorization list. Contains the null value if the privilege is not granted through an authorization list.
SYSTEM_SCHEMA_NAME	SYS_NAME	CHAR(10)	System name of the schema

SYSSCHEMAS

The SYSSCHEMAS view contains one row for every schema in the relational database.

For information related to a single schema, a query that uses table function OBJECT_STATISTICS will perform much better than querying SYSSCHEMAS. For example:

```
SELECT *
   FROM TABLE (QSYS2.OBJECT_STATISTICS('MJATST    ','LIB    ')) AS A
```

The following table describes the columns in the SYSSCHEMAS view:

Table 185. SYSSCHEMAS view

Column Name	System Column Name	Data Type	Description
SCHEMA_NAME	NAME	VARCHAR(128)	Name of the SQL schema.
SCHEMA_OWNER	OWNER	VARCHAR(128)	Owner of the schema.
SCHEMA_CREATOR	CREATOR	VARCHAR(128)	Name of the user that created the schema.
CREATION_TIMESTAMP	TIMESTAMP	TIMESTAMP	Timestamp when the schema was created.
SCHEMA_SIZE	SIZE	DECIMAL(15,0)	Size of the schema (in bytes).
SCHEMA_TEXT	LABEL	VARCHAR(50) Nullable	A character string supplied with the LABEL statement. Contains the null value if the schema has no text.
SYSTEM_SCHEMA_NAME	SYS_NAME	CHAR(10)	System schema name.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSSEQUENCEAUTH

The SYSSEQUENCEAUTH view contains one row for every privilege granted on a sequence. Note that this catalog view cannot be used to determine whether a user is authorized to a sequence because the privilege to use a sequence could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the SYSSEQUENCEAUTH view:

Table 186. SYSSEQUENCEAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
SEQUENCE_SCHEMA	SEQSCHEMA	VARCHAR(128)	Name of the schema
SEQUENCE_NAME	SEQNAME	VARCHAR(128)	Name of the sequence
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the sequence. **USAGE** The privilege to use the sequence.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	If the privilege is granted through an authorization, contains the name of the authorization list. Contains the null value if the privilege is not granted through an authorization list.
SYSTEM_SEQ_SCHEMA	SYSSSCHEMA	CHAR(10)	System name of the schema
SYSTEM_SEQ_NAME	SYSSNAME	CHAR(10)	System name of the sequence

SYSSEQUENCES

The SYSSEQUENCES view contains one row for every sequence object in the SQL schema.

The following table describes the columns in the SYSSEQUENCES view:

Table 187. SYSSEQUENCES view

Column name	System Column Name	Data Type	Description
SEQUENCE_SCHEMA	SEQSCHEMA	VARCHAR(128)	The name of the SQL schema containing the sequence.
SEQUENCE_NAME	SEQNAME	VARCHAR(128)	Name of the sequence.
MAXIMUM_VALUE	MAXVALUE	DECIMAL(63,0)	Maximum value of the sequence.
MINIMUM_VALUE	MINVALUE	DECIMAL(63,0)	Minimum value of the sequence.
INCREMENT	INCREMENT	INTEGER	Increment value of the sequence.
CYCLE_OPTION	CYCLE	VARCHAR(3)	Identifies whether the sequence values will continue to be generated after the minimum or maximum value has been reached. **NO** Values will not continue to be generated. **YES** Values will continue to be generated.
CACHE	CACHE	INTEGER	Specifies the number of sequence values that may be preallocated for faster access. Zero indicates that the values will not be preallocated.
ORDER	ORDER	VARCHAR(3)	Specifies whether the sequence values must be generated in order of the request. **NO** Values do not need to be generated in order of the request. **YES** Values must be generated in order of the request.
DATA_TYPE	DATA_TYPE	VARCHAR(128)	Type of sequence: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **DISTINCT** Distinct type
NUMERIC_PRECISION	PRECISION	INTEGER	The precision of all numeric columns.
USER_DEFINED_TYPE_SCHEMA	TYPESCHEMA	VARCHAR(128)	The name of the schema if this is a distinct type.
		Nullable	Contains the null value if the sequence is not a distinct type.
USER_DEFINED_TYPE_NAME	TYPENAME	VARCHAR(128)	The name of the distinct type.
		Nullable	Contains the null value if the sequence is not a distinct type.
START	START	DECIMAL(63,0)	Starting value of the sequence.
MAXASSIGNEDVAL	MAXASNVAL	DECIMAL(63,0)	Last possible assigned sequence value. This value includes any values that were cached, but not used.
		Nullable	Contains the null value when the sequence is created. Is not null after the first value is assigned.

Table 187. SYSSEQUENCES view (continued)

Column name	System Column Name	Data Type	Description
SEQUENCE_DEFINER	DEFINER	VARCHAR(128)	The authorization ID under which the sequence was created.
SEQUENCE_CREATED	CREATEDTS	TIMESTAMP	Timestamp when the sequence was created.
LAST_ALTERED_TIMESTAMP	ALTEREDTS	TIMESTAMP	Timestamp when the sequence was last altered.
SEQUENCE_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200	A character string supplied with the LABEL statement (sequence text).
		Nullable	Contains the null value if the sequence has no sequence text.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement.
		Nullable	Contains the null value if there is no long comment.
SYSTEM_SEQ_SCHEMA	SYSSSCHEMA	CHAR(10)	The system name of the schema
SYSTEM_SEQ_NAME	SYSSNAME	CHAR(10)	The system name of the sequence
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSTABAUTH

The SYSTABAUTH view contains one row for every privilege granted on a table or view. Note that this catalog view cannot be used to determine whether a user is authorized to a table or view because the privilege to use a table or view could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the SYSTABAUTH view:

Table 188. SYSTABAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
TABLE_SCHEMA	DBNAME	VARCHAR(128)	Name of the schema
TABLE_NAME	NAME	VARCHAR(128)	Name of the table
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the table. **DELETE** The privilege to delete rows from the table. **INDEX** The privilege to create an index on the table. **INSERT** The privilege to insert rows into the table. **REFERENCES** The privilege to reference the table in a referential constraint. **SELECT** The privilege to select rows from the table. **UPDATE** The privilege to update the table.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	If the privilege is granted through an authorization, contains the name of the authorization list. Contains the null value if the privilege is not granted through an authorization list.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table or view

SYSTABLEDEP

The SYSTABLEDEP view records the dependencies of materialized query tables.

The following table describes the columns in the SYSTABLEDEP view:

Table 189. SYSTABLEDEP view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table, view or alias
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table, view or alias. This is the SQL table, view or alias name if it exists; otherwise, it is the system table, view or alias name.
OBJECT_SCHEMA	BSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the object.
OBJECT_NAME	BNAME	VARCHAR(128)	Name of the object the materialized query table is dependent on.
OBJECT_TYPE	BTYPE	CHAR(24)	Indicates the object type of the object referenced in the materialized query table: **FUNCTION** The object is a function. **TABLE** The object is a table. **TYPE** The object is a distinct type. **VIEW** The object is a view.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number of the object.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
PARM_SIGNATURE	SIGNATURE	VARCHAR(10000)	This column identifies the routine signature.
		Nullable	Contains the null value if the object is not a routine.

SYSTABLEINDEXSTAT

The SYSTABLEINDEXSTAT view contains one row for every index that has at least one partition or member built over a table. If the index is over more than one partition or member, the statistics include all those partitions and members. If the table is a distributed table, the partitions that reside on other database nodes are not included. They are contained in the catalog views of the other database nodes.

The following table describes the columns in the SYSTABLEINDEXSTAT view:

Table 190. SYSTABLEINDEXSTAT view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
PARTITION_TYPE	PARTTYPE	CHAR(1)	The type of the table partitioning: **blank** The table is not partitioned. **H** This is data hash partitioning. **R** This is data range partitioning. **D** This is distributed database hash partitioning.
NUMBER_PARTITIONS	NBRPARTS	INTEGER	Number of partitions or members of the table.
NUMBER_DISTRIBUTED_PARTITIONS	DSTPARTS	INTEGER Nullable	If the table is a distributed table, contains the total number of partitions. If the table is not a distributed table, contains null.
INDEX_SCHEMA	INDSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the index, logical file, or constraint.
INDEX_NAME	INDNAME	VARCHAR(128)	Name of the index, logical file, or constraint.
INDEX_TYPE	INDTYPE	VARCHAR(11)	The type of the index: **INDEX** The index is an SQL index. **LOGICAL** The index is part of a logical file. **PRIMARY KEY** The index is a primary key constraint. **UNIQUE** The index is a unique constraint. **REFERENTIAL** The index is a foreign key constraint.
NUMBER_KEY_COLUMNS	INDKEYS	BIGINT	Number of columns that define the index key.
COLUMN_NAMES	COLNAMES	VARCHAR(1024)	A comma separated list of column names that define the index key. If the length of all the column names exceeds 1024, '...' is returned at the end of the column value.
NUMBER_LEAF_PAGES	NLEAF	BIGINT	Not applicable for DB2 for i. Will always be -1.
NUMBER_LEVELS	NLEVELS	SMALLINT	Not applicable for DB2 for i. Will always be -1.
FIRSTKEYCARD	KEYCARD1	BIGINT	The total number of distinct first key values for all index partitions. If the index is an encoded vector index, this is the total number of unique values for the entire index key.
FIRST2KEYCARD	KEYCARD2	BIGINT	The total number of distinct keys using the first two columns for all index partitions. If the index is an encoded vector index, -1 is returned.
FIRST3KEYCARD	KEYCARD3	BIGINT	The total number of distinct keys using the first three columns for all index partitions. If the index is an encoded vector index, -1 is returned.
FIRST4KEYCARD	KEYCARD4	BIGINT	The total number of distinct keys using the first four columns for all index partitions. If the index is an encoded vector index, -1 is returned.

Table 190. SYSTABLEINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
FULLKEYCARD	KEYCARDF	BIGINT	The total number of distinct full key values for all index partitions. If the index has more than 4 key columns or is an encoded vector index, -1 is returned.
CLUSTERRATIO	CLSRATIO	SMALLINT	Not applicable for DB2 for i. Will always be -1.
CLUSTERFACTOR	CLSFACTOR	DOUBLE	Not applicable for DB2 for i. Will always be -1.
SEQUENTIAL_PAGES	SEQPAGES	BIGINT	Not applicable for DB2 for i. Will always be -1.
DENSITY	DENSITY	INTEGER	Not applicable for DB2 for i. Will always be -1.
PAGE_FETCH_PAIRS	FETCHPAIRS	VARCHAR(520)	Not applicable for DB2 for i. Will always be -1.
NUMBER_KEYS	NUMRIDS	BIGINT	The total number of keys for all index partitions. If the index is invalid or is an encoded vector index, -1 is returned.
NUMRIDS_DELETED	NUMRIDSDLT	BIGINT	Not applicable for DB2 for i. Will always be 0.
NUM_EMPTY_LEAFS	EMPTYLEAFS	BIGINT	Not applicable for DB2 for i. Will always be 0.
AVERAGE_RANDOM_FETCH_PAGES	AVGRNDFTCH	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_RANDOM_PAGES	AVGRNDPAGE	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_SEQUENCE_GAP	AVGSEQGAP	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_SEQUENCE_FETCH_GAP	AVGSEQFGAP	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_SEQUENCE_PAGES	AVGSEQPAGE	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVERAGE_SEQUENCE_FETCH_PAGES	AVGSEQFPAG	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVGPARTITION_CLUSTERRATIO	PCLSRATIO	SMALLINT	Not applicable for DB2 for i. Will always be -1.
AVGPARTITION_CLUSTERFACTOR	PCLSFACTOR	DOUBLE	Not applicable for DB2 for i. Will always be -1.
AVGPARTITION_PAGE_FETCH_PAIRS	PFETCHPAIR	VARCHAR(520)	Not applicable for DB2 for i. Will always be an empty string.
DATAPARTITION_CLUSTERFACTOR	DCLSFACTOR	DOUBLE	A statistic measuring the "clustering" of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.
INDCARD	INDCARD	BIGINT	Number of keys in the index. If the index is invalid or is an encoded vector index, -1 is returned.
INDEX_VALID	VALID	CHAR(1)	An indication or whether any index is invalid and needs to be rebuilt: **0** At least one partition or member for the index is invalid. **1** All partitions or members for the index are valid.
INDEX_HELD	HELD	CHAR(1)	An indication or whether a pending rebuild of the index is currently held by the user: **0** No rebuilds are pending or held for any partition or member of the index. **1** A pending rebuild for at least one partition or member for the index is held.
CREATE_TIMESTAMP	CREATED	TIMESTAMP	Maximum timestamp when any partition or member of the index was created.
LAST_BUILD_TIMESTAMP	LASTBUILD	TIMESTAMP	Maximum timestamp when any partition or member of the index was last rebuilt.
LAST_QUERY_USE	LASTQRYUSE	TIMESTAMP Nullable	Maximum timestamp of the last time any partition or member of the index was used in a query since the last time the usage statistics were reset. If no partition or member of the index has ever been used in a query since the last time the usage statistics were reset, contains null.

Table 190. SYSTABLEINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
LAST_STATISTICS_USE	LASTSTUSE	TIMESTAMP Nullable	Maximum timestamp of the last time any partition or member of the index was used by the optimizer for statistics since the last time the usage statistics were reset. If no partition or member of the index has ever been used for statistics since the last time the usage statistics were reset, contains null.
QUERY_USE_COUNT	QRYUSECNT	BIGINT	Total number of times any partition or member of the index was used in a query since the last time the usage statistics were reset. If no partition or member of the index has ever been used in a query since the last time the usage statistics were reset, contains 0.
QUERY_STATISTICS_COUNT	QRYSTCNT	BIGINT	Total number of times any partition or member of the index was used by the optimizer for statistics since the last time the usage statistics were reset. If no partition or member of the index has ever been used for statistics since the last time the usage statistics were reset, contains 0.
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP Nullable	Maximum timestamp of the last time any partition or member of the index was used directly by an application for native record I/O or SQL operations. If no partition or member of the index has ever been used, contains null.
DAYS_USED_COUNT	DAYSUSED	INTEGER	Maximum number of days any partition or member of the index was used directly by an application for native record I/O or SQL operations since the last time the usage statistics were reset. If no partition or member of the index has ever been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	Maximum timestamp of the last time the usage statistics were reset for the index. For more information see the Change Object Description (CHGOBJD) command. If the index's last used timestamp has never been reset, contains null.
INDEX_SIZE	SIZE	BIGINT	Total size (in bytes) of the binary trees or encoded vector indexes of all partitions or members of the index.
ESTIMATED_BUILD_TIME	ESTBLDTIME	INTEGER	Maximum estimated time (in seconds) required to rebuild any partition or member of the index.
LAST_BUILD_TIME	LSTBLDTIME	INTEGER Nullable	Elapsed time (in seconds) the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_KEYS	LSTBLDKEYS	BIGINT Nullable	Number of keys the last time the index was built. Contains null if the last build information is not available.
LAST_BUILD_DEGREE	LSTBLDDEG	SMALLINT Nullable	Parallel degree the last time the index was built. Contains null if the last build information is not available.
DELAYED_MAINT_KEYS	DLYKEYS	INTEGER Nullable	Maximum number of keys that need to be inserted into the binary tree of any partition or member of a delayed maintenance index. If the index is not a delayed maintenance index, contains null.
SPARSE	SPARSE	CHAR(1)	Indicates whether the index contains keys for all the rows of its depended on table: **0** The index contains keys for all the rows of its depended on table. **1** The index is a select/omit logical file and does not contain keys for all the rows of its depended on table.

Table 190. SYSTABLEINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
DERIVED_KEY	DERIVED	CHAR(1)	Indicates whether the any key columns in the index are expressions: **0** No key columns of the index are expressions. **1** At least one key column is an expression. Currently, this is only possible in a DDS-created logical file or temporary index.
PARTITIONED	PARTITION	CHAR(1)	Indicates whether the index is partitioned or not partitioned: **0** An SQL index is not partitioned (spans multiple partitions). **1** The index is not built over a partitioned table or built over a partitioned table and is partitioned (does not span multiple partitions or members). **2** The index is a logical file built over multiple partitions or members.
ACCPTH_TYPE	ACCPTHTYPE	CHAR(1)	Indicates the type of index: **0** The index is a maximum 1 terabyte (*MAX1TB) binary radix index. **1** The index is a maximum 4 gigabyte (*MAX4GB) binary radix index. **2** The index is an encoded vector index.
UNIQUE	UNIQUE	CHAR(1)	Indicates whether an index is unique: **0** The index is a UNIQUE index. **1** The index is a UNIQUE WHERE NOT NULL index. **2** The index is a non-unique first-in-first-out (FIFO) index. **3** The index is a non-unique last-in-last-out (LIFO) index. **4** The index is a non-unique first-change-first-out (FCFO) index.
SRTSEQ_TYPE	SRTSEQ	CHAR(1)	Indicates whether the index uses a collating sequence: **0** The index does not use a collating table. **1** The index uses an alternate collating sequence (ALTSEQ). **2** The index uses a sort sequence (SRTSEQ).
LOGICAL_PAGE_SIZE	PAGE_SIZE	INTEGER Nullable	The logical page size of the index. If the index is an encoded vector index, contains null.
OVERFLOW_VALUES	OVERFLOW	INTEGER Nullable	Maximum number of distinct key values that have overflowed any partition or member of the encoded vector index. If the index is not an encoded vector index, contains null.
EVI_CODE_SIZE	CODE_SIZE	INTEGER Nullable	The size of the byte code of the encoded vector index. If the index is not an encoded vector index, contains null.
LOGICAL_READS	LGLREADS	BIGINT	Total number of logical read operations for any partition or member of the index since the last IPL.
PHYSICAL_READS	PHYREADS	BIGINT	Not applicable for DB2 for i. Will always be 0.

Table 190. SYSTABLEINDEXSTAT view (continued)

Column name	System Column Name	Data Type	Description
SEQUENTIAL_READS	SEQREADS	BIGINT	Number of sequential read operations for the index since the last IPL.
RANDOM_READS	RANREADS	BIGINT	Number of random read operations for the index since the last IPL.
SEARCH_CONDITION	IXWHERECON	VARGRAPHIC(1024) CCSID 1200	If an index is sparse, the search condition of the index. If the length of the search condition exceeds 1024, '...' is returned at the end of the column value.
KEEP_IN_MEMORY	KEEPINMEM	CHAR(1)	Indicates whether the index should be kept in memory: **0** No memory preference. **1** The index should be kept in memory, if possible.
MEDIA_PREFERENCE	MEDIAPREF	SMALLINT	Indicates the media preference of the index: **0** No media preference. **255** The index should be allocated on Solid State Disk (SSD), if possible.
INCLUDE_EXPRESSION	IXINCEXPR	VARGRAPHIC(1024) CCSID 1200 Nullable	Index INCLUDE expression. Contains null if the index does not have an INCLUDE expression.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.

SYSTABLES

The SYSTABLES view contains one row for every table, view or alias in the SQL schema, including the tables and views of the SQL catalog.

The following table describes the columns in the SYSTABLES view:

Table 191. SYSTABLES view

Column name	System Column Name	Data Type	Description
TABLE_NAME	NAME	VARCHAR(128)	Name of the table, view or alias. This is the SQL table, view or alias name if it exists; otherwise, it is the system table, view or alias name.
TABLE_OWNER	CREATOR	VARCHAR(128)	Owner of the table, view or alias
TABLE_TYPE	TYPE	CHAR(1)	If the row describes a table, view, or alias: **A** Alias **L** Logical file **M** Materialized query table **P** Physical file **T** Table **V** View
COLUMN_COUNT	COLCOUNT	INTEGER	Number of columns in the table or view. Zero for an alias.
ROW_LENGTH	RECLENGTH [152]	INTEGER	Maximum length of any record in the table. Zero for an alias.
TABLE_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200	A character string provided with the LABEL statement.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement.
		Nullable	Contains the null value if there is no long comment.
TABLE_SCHEMA	DBNAME	VARCHAR(128)	Name of the SQL schema that contains the table, view or alias
LAST_ALTERED_TIMESTAMP	ALTEREDTS	TIMESTAMP	Timestamp when the table was last altered or created.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name
FILE_TYPE	FILETYPE	CHAR(1)	File type **D** Data file or alias **S** Source file
BASE_TABLE_CATALOG	LOCATION	VARCHAR(18)	For an alias, this is the name of the relational database that contains the table or view the alias is based on.
		Nullable	Contains the null value if the table is not an alias.
BASE_TABLE_SCHEMA	TBDBNAME	VARCHAR(128)	For an alias, this is the name of the SQL schema that contains the table or view the alias is based on.
		Nullable	Contains the null value if the table is not an alias.
BASE_TABLE_NAME	TBNAME	VARCHAR(128)	For an alias, this is the name of the table or view the alias is based on.
		Nullable	Contains the null value if the table is not an alias.
BASE_TABLE_MEMBER	TBMEMBER	VARCHAR(10)	For an alias, this is the name of the file member the alias is based on. Contains *FIRST if this is an alias, but a member name was not specified.
		Nullable	Contains the null value if the table is not an alias.

Table 191. SYSTABLES view (continued)

Column name	System Column Name	Data Type	Description
SYSTEM_TABLE	SYSTABLE	CHAR(1)	System table **N** The table is not a system table. **Y** The table is a system table.
SELECT_OMIT	SELECTOMIT	CHAR(1)	Select/omit logical file **D** The table is a dynamic select/omit logical file. **N** The table is not a select/omit logical file. **Y** The table is a select/omit logical file.
IS_INSERTABLE_INTO	INSERTABLE	VARCHAR(3)	Identifies whether an INSERT is allowed on the table. **NO** An INSERT is not allowed on this table. **YES** An INSERT is allowed on this table.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
ENABLED	ENABLED	VARCHAR(3) Nullable	Indicates whether the materialized query table is enabled for optimization: **NO** The materialized query table is not enabled for optmization. **YES** The materialized query table is enabled for optmization. Contains the null value if the table is not a materialized query table.
MAINTENANCE	MAINTAIN	VARCHAR(6) Nullable	Indicates whether the materialized query table is user or system maintained: **USER** The materialized query table is user maintained. Contains the null value if the table is not a materialized query table.
REFRESH	REFRESH	VARCHAR(9) Nullable	Indicates the materialized query table REFRESH option: **DEFERRED** The materialized query table is REFRESH DEFERRED. Contains the null value if the table is not a materialized query table.
REFRESH_TIME	REFRESHDTS	TIMESTAMP Nullable	Indicates the timestamp of the last materialized query table REFRESH: Contains the null value if the table is not a materialized query table or if the table has never been refreshed.
MQT_DEFINITION	MQTDEF	DBCLOB(2M) 13488 CCSID Nullable	Indicates the query expression of the materialized query table: Contains the null value if the table is not a materialized query table.

Table 191. SYSTABLES view (continued)

Column name	System Column Name	Data Type	Description
ISOLATION	ISOLATION	CHAR(2) Nullable	Indicates the isolation level used for the *select-statement* when refreshing the materialized query table: **RR** Repeatable Read (*RR) **RS** Read Stability (*ALL) **CS** Cursor Stability (*CS) **UR** Uncommitted Read (*CHG) **NC** No Commit (*NONE) Contains the null value if the table is not a materialized query table.
PARTITION_TABLE	PART_TABLE	VARCHARA(11)	Indicates whether the table is a partitioned table: **DISTRIBUTED** The table is a distributed table. **NO** The table is not a partitioned table. **YES** The table is a partitioned table.
TABLE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the table.
MQT_RESTORE_DEFERRED	MQTRSTDFR	CHAR(1)	If the table is a materialized query table: **Y** The MQT is deferred as the result of a restore. **N** The MQT is not deferred. Contains the null value if the table is not a materialized query table.
ROUNDING_MODE	DECFLTRND	CHAR(1)	Indicates the DECFLOAT rounding mode of the materialized query table or view: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP Contains the null value if the table is not a view or MQT, or if the materialized query table or view does not have an expression that references a DECFLOAT column, function, or constant.

152. The length is the number of bytes passed in database buffers, not the internal storage length.

SYSTABLESTAT

The SYSTABLESTAT view contains one row for every table that has at least one partition or member. If the table has more than one partition or member, the statistics include all partitions and members. If the table is a distributed table, the partitions that reside on other database nodes are not included. They are contained in the catalog views of the other database nodes.

The following table describes the columns in the SYSTABLESTAT view:

Table 192. SYSTABLESTAT view

Column name	System Column Name	Data Type	Description
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table.
PARTITION_TYPE	PARTTYPE	CHAR(1)	The type of the table partitioning: **blank** The table is not partitioned. **H** This is data hash partitioning. **R** This is data range partitioning. **D** This is distributed database hash partitioning.
NUMBER_PARTITIONS	NBRPARTS	INTEGER	Number of partitions or members of the table.
NUMBER_DISTRIBUTED_PARTITIONS	DSTPARTS	INTEGER Nullable	If the table is a distributed table, contains the total number of partitions. If the table is not a distributed table, contains null.
NUMBER_ROWS	CARD	BIGINT	Number of valid rows in all partitions or members of the table.
NUMBER_ROW_PAGES	NPAGES	BIGINT	Number of 64K pages in all partitions or members of the table.
NUMBER_PAGES	FPAGES	BIGINT	Same as NUMBER_ROW_PAGES.
OVERFLOW	OVERFLOW	BIGINT	The estimated number of rows that have overflowed to variable length segments. If the table does not contain variable length or LOB columns, contains 0.
CLUSTERED	CLUSTERED	CHAR(1) Nullable	Not applicable for DB2 for i. Will always be null.
ACTIVE_BLOCKS	ACTBLOCKS	BIGINT	Not applicable for DB2 for i. Will always be -1.
AVGCOMPRESSEDROWSIZE	ACROWSIZE	BIGINT	Not applicable for DB2 for i. Will always be -1.
AVGROWCOMPRESSIONRATIO	ACROWRATIO	REAL	Not applicable for DB2 for i. Will always be -1.
AVGROWSIZE	AVGROWSIZE	BIGINT	Average length (in bytes) of a row in this table. If the table has variable length or LOB columns, contains -1.
PCTROWSCOMPRESSED	PCTCROWS	REAL	Not applicable for DB2 for i. Will always be -1.
PCTPAGESSAVED	PCTPGSAVED	SMALLINT	Not applicable for DB2 for i. Will always be -1.
NUMBER_DELETED_ROWS	DELETED	BIGINT	Number of deleted rows in all partitions or members of the table.
DATA_SIZE	SIZE	BIGINT	Total size (in bytes) of the data spaces in all partitions or members of the table.
VARIABLE_LENGTH_SIZE	VLSIZE	BIGINT	Size (in bytes) of the variable-length data space segments in all partitions or members of the table.
FIXED_LENGTH_EXTENTS	FLEXTENTS	BIGINT	Number of fixed-length data space segment extents in all partitions or members of the table.
VARIABLE_LENGTH_EXTENTS	VLEXTENTS	BIGINT	Number of variable-length data space segment extents in all partitions or members of the table.
COLUMN_STATS_SIZE	CSTATSSIZE	BIGINT	Size (in bytes) of the column statistics in all partitions or members of the table.
MAINTAINED_TEMPORARY_INDEX_SIZE	MTISIZE	BIGINT	Size (in bytes) of all maintained temporary indexes over any partitions or members of the table.

Table 192. SYSTABLESTAT view (continued)

Column name	System Column Name	Data Type	Description
NUMBER_DISTINCT_INDEXES	DISTINCTIX	INTEGER	The number of distinct indexes built over any partitions or members of the table. This does not include maintained temporary indexes.
OPEN_OPERATIONS	OPENS	BIGINT	Number of full opens of all partitions or members of the table since the last IPL.
CLOSE_OPERATIONS	CLOSES	BIGINT	Number of full closes of all partitions or members of the table since the last IPL.
INSERT_OPERATIONS	INSERTS	BIGINT	Number of insert operations of all partitions or members of the table since the last IPL.
UPDATE_OPERATIONS	UPDATES	BIGINT	Number of update operations of all partitions or members of the table since the last IPL.
DELETE_OPERATIONS	DELETES	BIGINT	Number of delete operations of all partitions or members of the table since the last IPL.
CLEAR_OPERATIONS	DSCLEARS	BIGINT	Number of clear operations (CLRPFM operations) of all partitions or members of the table since the last IPL.
COPY_OPERATIONS	DSCOPIES	BIGINT	Number of data space copy operations (certain CPYxxx operations) of all partitions or members of the table since the last IPL.
REORGANIZE_OPERATIONS	DSREORGS	BIGINT	Number of data space reorganize operations (non-interruptible RGZPFM operations) of all partitions or members of the table since the last IPL.
INDEX_BUILDS	DSINXBLDS	BIGINT	Number of creates or rebuilds of indexes that reference any partition or member of the table since the last IPL. This does not include maintained temporary indexes.
LOGICAL_READS	LGLREADS	BIGINT	Number of logical read operations of all partitions or members of the table since the last IPL.
PHYSICAL_READS	PHYREADS	BIGINT	Number of physical read operations of all partitions or members of the table since the last IPL.
SEQUENTIAL_READS	SEQREADS	BIGINT	Number of sequential read operations of all partitions or members of the table since the last IPL.
RANDOM_READS	RANREADS	BIGINT	Number of random read operations of all partitions or members of the table since the last IPL.
LAST_CHANGE_TIMESTAMP	LASTCHG	TIMESTAMP	Maximum timestamp of the last change that occurred to any partition or member of the table.
LAST_SAVE_TIMESTAMP	LASTSAVE	TIMESTAMP Nullable	Minimum timestamp of the last save of any partition or member of the table. If no partition or member has been saved, contains null.
LAST_RESTORE_TIMESTAMP	LASTRST	TIMESTAMP Nullable	Maximum timestamp of the last restore any partition or member of the table. If no partition or member has been restored, contains null.
LAST_USED_TIMESTAMP	LASTUSED	TIMESTAMP Nullable	Maximum timestamp of the last time any partition or member was used directly by an application for native record I/O or SQL operations. If no partition or member has ever been used, contains null.
DAYS_USED_COUNT	DAYSUSED	INTEGER	Maximum number of days any partition or member was used directly by an application for native record I/O or SQL operations since the last time the usage statistics were reset. If no partition or member has been used since the last time the usage statistics were reset, contains 0.
LAST_RESET_TIMESTAMP	LASTRESET	TIMESTAMP Nullable	Maximum timestamp of the last time the usage statistics were reset for the table. For more information see the Change Object Description (CHGOBJD) command. If no partition or member's last used timestamp has ever been reset, contains null.

Table 192. SYSTABLESTAT view (continued)

Column name	System Column Name	Data Type	Description
NUMBER_PARTITIONING_KEYS	NBRPKEYS	INTEGER	The number of partitioning keys. If the table is not partitioned, contains null.
		Nullable	
PARTITIONING_KEYS	PARTKEYS	VARCHAR(2880)	The list of partitioning keys. If the table is not partitioned, contains null.
		Nullable	
VOLATILE	VOLATILE	CHAR(1)	Indicates whether the table is volatile. If the table is not an SQL table or physical file, contains null.
		Nullable	
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.

SYSTRIGCOL

The SYSTRIGCOL view contains one row for each column either implicitly or explicitly referenced in the WHEN clause or the triggered SQL statements of a trigger.

The following table describes the columns in the SYSTRIGCOL view:

Table 193. SYSTRIGCOL view

Column Name	System Column Name	Data Type	Description
TRIGGER_SCHEMA	TRIGSCHEMA	VARCHAR(128)	Name of the schema containing the trigger.
TRIGGER_NAME	TRIGNAME	VARCHAR(128)	Name of the trigger.
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the schema containing the table or view that contains the column that is referenced in the trigger.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table or view that contains the column that is referenced in the trigger.
COLUMN_NAME	TABCOLUMN	VARCHAR(128)	Name of the column that is referenced in the trigger.
OBJECT_TYPE	BTYPE	CHAR(24)	Indicates the object type of the object that contains the column referenced in the trigger: **FUNCTION** The object is a function. **MATERIALIZED QUERY TABLE** The object is a materialized query table. **TABLE** The object is a table. **VIEW** The object is a view.
SYSTEM_TRIGGER_SCHEMA	SYS_TDNAME	CHAR(10)	System schema name.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10) Nullable	System schema name of the table or view that contains the column that is referenced in the trigger. Contains the null value if the referenced table does not exist or if the referenced object is a table function.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10) Nullable	System table name of the table or view that contains the column that is referenced in the trigger. Contains the null value if the referenced table does not exist or if the referenced object is a table function.

SYSTRIGDEP

The SYSTRIGDEP view contains one row for each object referenced in the WHEN clause or the triggered SQL statements of a trigger.

The following table describes the columns in the SYSTRIGDEP view:

Table 194. SYSTRIGDEP view

Column Name	System Column Name	Data Type	Description
TRIGGER_SCHEMA	TRIGSCHEMA	VARCHAR(128)	Name of the schema containing the trigger.
TRIGGER_NAME	TRIGNAME	VARCHAR(128)	Name of the trigger.
OBJECT_SCHEMA	BSCHEMA	VARCHAR(128)	Name of the schema containing the object referenced in the trigger.
OBJECT_NAME	BNAME	VARCHAR(128)	Name of the object referenced in the trigger.
OBJECT_TYPE	BTYPE	CHAR(24)	Indicates the object type of the object referenced in the trigger: **ALIAS** The object is an alias. **FUNCTION** The object is a function. **INDEX** The object is an index. **MATERIALIZED QUERY TABLE** The object is a materialized query table. **PACKAGE** The object is a package. **PROCEDURE** The object is a procedure. **SCHEMA** The object is a schema. **SEQUENCE** The object is a sequence. **TABLE** The object is a table. **TYPE** The object is a distinct type. **VARIABLE** The object is a variable. **VIEW** The object is a view.
PARM_SIGNATURE	SIGNATURE	VARCHAR(10000) Nullable	This column identifies the routine signature. Contains the null value if the object is not a routine.
SYSTEM_TRIGGER_SCHEMA	SYS_TDNAME	CHAR(10)	System schema name.
SYSTEM_OBJECT_SCHEMA	SYS_DNAME	CHAR(10) Nullable	System schema name of the object referenced in the trigger. Contains the null value if the referenced object does not exist or if the referenced object is not an alias, index, materialized query table, table, or view.
SYSTEM_OBJECT_NAME	SYS_ONAME	CHAR(10) Nullable	System object name of the object referenced in the trigger. Contains the null value if the referenced object does not exist or if the referenced object is not an alias, index, materialized query table, table, or view.

SYSTRIGGERS

The SYSTRIGGERS view contains one row for each trigger in an SQL schema.

The following table describes the columns in the SYSTRIGGERS view:

Table 195. SYSTRIGGERS view

Column Name	System Column Name	Data Type	Description
TRIGGER_SCHEMA	TRIGSCHEMA	VARCHAR(128)	Name of the schema containing the trigger.
TRIGGER_NAME	TRIGNAME	VARCHAR(128)	Name of the trigger.
EVENT_MANIPULATION	TRIGEVENT	VARCHAR(6)	Indicates the event that causes the trigger to fire: **DELETE** Trigger fires on a DELETE. **INSERT** Trigger fires on a INSERT. **UPDATE** Trigger fires on a DELETE. **READ** Trigger fires when a row is read. This is only valid for triggers created via the ADDPFTRG command. **MULTI** Trigger fires for multiple events. The EVENTUPDATE, EVENTDELETE, and EVENTINSERT columns define the events.
EVENT_OBJECT_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the schema containing the subject table or view of the trigger.
EVENT_OBJECT_TABLE	TABNAME	VARCHAR(128)	Name of the subject table or view of the trigger.
ACTION_ORDER	ORDERSEQNO	INTEGER	The ordinal position of this trigger in the list of triggers for the table or view. This indicates the order in which the trigger will be fired.
ACTION_CONDITION	CONDITION	DBCLOB(2097152) CCSID 13488 Nullable	Text of the WHEN clause for the trigger. Contains the null value if there is no WHEN clause.
ACTION_STATEMENT	TEXT	DBCLOB(2097152) CCSID 13488 Nullable	Text of the SQL statements in the trigger action. Contains the null value if this is a trigger created via the ADDPFTRG command.
ACTION_ORIENTATION	GRANULAR	VARCHAR(9)	Indicates whether this is a ROW or STATEMENT trigger: **ROW** Trigger fires for each ROW. **STATEMENT** Trigger fires for each statement.
ACTION_TIMING	TRIGTIME	VARCHAR(7)	Indicates whether this is a BEFORE, AFTER, or INSTEAD OF trigger: **BEFORE** Trigger fires before the triggering event. **AFTER** Trigger fires after the triggering event. **INSTEAD** Trigger fires instead of the triggering event.
TRIGGER_MODE	TRIGMODE	VARCHAR(6)	Indicates the firing mode for the trigger: **DB2SQL** The trigger mode is DB2SQL. **DB2ROW** The trigger mode is DB2ROW.
ACTION_REFERENCE_OLD_ROW	OLD_ROW	VARCHAR(128) Nullable	Name of the OLD ROW correlation name. Contains the null value if an OLD ROW correlation name was not specified.
ACTION_REFERENCE_NEW_ROW	NEW_ROW	VARCHAR(128) Nullable	Name of the NEW ROW correlation name. Contains the null value if a NEW ROW correlation name was not specified.

Table 195. SYSTRIGGERS view (continued)

Column Name	System Column Name	Data Type	Description
ACTION_REFERENCE_OLD_TABLE	OLD_TABLE	VARCHAR(128)	Name of the OLD TABLE correlation name.
		Nullable	Contains the null value if an OLD TABLE correlation name was not specified.
ACTION_REFERENCE_NEW_TABLE	NEW_TABLE	VARCHAR(128)	Name of the NEW TABLE correlation name.
		Nullable	Contains the null value if a NEW TABLE correlation name was not specified.
SQL_PATH	SQL_PATH	VARCHAR(3483)	SQL path used when the trigger was created.
		Nullable	Contains the null value if the trigger was created via the ADDPFTRG command.
CREATED	CREATE_DTS	TIMESTAMP	Timestamp when the trigger was created.
TRIGGER_PROGRAM_NAME	TRIGPGM	VARCHAR(128)	Name of the trigger program.
TRIGGER_PROGRAM_LIBRARY	TRIGPGMLIB	VARCHAR(128)	System name of the schema containing the trigger program.
OPERATIVE	OPERATIVE	VARCHAR(1)	Indicates whether the trigger is operative. A table or view that has a trigger that contains a reference to that same table or view in its *triggered–action* is self-referencing. If a self-referencing trigger is duplicated into another library, restored into another library, moved into another library, or renamed; the trigger is marked inoperative since the table references in the *triggered–action* are unchanged and still reference the original schema and table name. **Y** The trigger is operative. **N** The trigger is inoperative.
ENABLED	ENABLED	VARCHAR(1)	Indicates whether the trigger is enabled (see the CL command CHGPFTRG) **Y** The trigger is enabled. **N** The trigger is disabled.
THREADSAFE	THDSAFE	VARCHAR(8)	Indicates whether the trigger is thread safe. **YES** The trigger is thread safe. **NO** The trigger is not thread safe. **UNKNOWN** The thread safety of the trigger is unknown.
MULTITHREADED_JOB_ACTION	MLTTHDACN	VARCHAR(8)	Indicates the action to take when the trigger program is called in a multithreaded job. **SYSVAL** Use the QMLTTHDACN system value to determine the action to take. **MSG** Run the trigger program in a multithreaded job, but send a diagnostic message. **NORUN** Do not run the trigger program in a multithreaded job. **RUN** Run the trigger program in a multithreaded job.
ALLOW_REPEATED_CHANGE	ALWREPCHG	VARCHAR(8)	Indicates the condition under which an update event fires the trigger. **YES** The trigger allows repeated changes to the same row. **NO** The trigger does not allow repeated changes to the same row.

Table 195. SYSTRIGGERS view (continued)

Column Name	System Column Name	Data Type	Description
TRIGGER_UPDATE_CONDITION	TRGUPDCND	CHAR(8) Nullable	Indicates whether an UPDATE trigger is always fired on an update event or only when a column value is actually changed. **ALWAYS** The trigger is always fired on an update event. **CHANGE** The trigger is only fired on an update event if a column value is actually changed. Contains the null value if the trigger is not an UPDATE trigger.
TRIGGER_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the trigger.
TRIGGER_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	A character string provided with the LABEL statement. Contains the null value if there is no label.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 13488 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
ROUNDING_MODE	DECFLTRND	CHAR(1) Nullable	The rounding mode for the trigger: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP Contains the null value if the trigger was created via the ADDPFTRG command.
SYSTEM_TRIGGER_SCHEMA	SYS_TDNAME	CHAR(10)	System schema name.
SYSTEM_EVENT_OBJECT_SCHEMA	SYS_DNAME	CHAR(10)	System schema name of the schema containing the subject table or view of the trigger.
SYSTEM_EVENT_OBJECT_TABLE	SYS_TNAME	CHAR(10)	System table name of the table or view that contains the subject table or view of the trigger.
EVENTUPDATE	EVENT_U	CHAR(1)	Indicates whether an update event fires this trigger. **Y** Yes **N** No
EVENTINSERT	EVENT_I	CHAR(1)	Indicates whether an insert event fires this trigger. **Y** Yes **N** No
EVENTDELETE	EVENT_D	CHAR(1)	Indicates whether a delete event fires this trigger. **Y** Yes **N** No

SYSTRIGUPD

The SYSTRIGUPD view contains one row for each column identified in the UPDATE column list, if any.

The following table describes the columns in the SYSTRIGUPD view:

Table 196. SYSTRIGUPD view

Column Name	System Column Name	Data Type	Description
TRIGGER_SCHEMA	TRIGSCHEMA	VARCHAR(128)	Name of the schema containing the trigger.
TRIGGER_NAME	TRIGNAME	VARCHAR(128)	Name of the trigger.
EVENT_OBJECT_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the schema containing the subject table of the trigger.
EVENT_OBJECT_TABLE	TABNAME	VARCHAR(128)	Name of the subject table of the trigger.
TRIGGERED_UPDATE_COLUMNS	TABCOLUMN	VARCHAR(128)	Name of a column specified in the UPDATE column list of the trigger.
SYSTEM_TRIGGER_SCHEMA	SYS_TDNAME	CHAR(10)	System schema name.
SYSTEM_EVENT_OBJECT_SCHEMA	SYS_DNAME	CHAR(10)	System schema name of the schema containing the subject table or view of the trigger.
SYSTEM_EVENT_OBJECT_TABLE	SYS_TNAME	CHAR(10)	System table name of the table or view that contains the subject table or view of the trigger.

SYSTYPES

The SYSTYPES table contains one row for each built-in data type and each distinct type and array type created by the CREATE TYPE statement.

The following table describes the columns in the SYSTYPES table:

Table 197. SYSTYPES table

Column Name	System Column Name	Data Type	Description
USER_DEFINED_TYPE_SCHEMA	TYPESCHEMA	VARCHAR(128)	Schema name of the data type.
USER_DEFINED_TYPE_NAME	TYPENAME	VARCHAR(128)	Name of the data type.
USER_DEFINED_TYPE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that created the data type.
SOURCE_SCHEMA	SRCSCHEMA	VARCHAR(128)	The schema for the source data type of this data type.
		Nullable	Contains the null value if this is a built-in data type.
SOURCE_TYPE	SRCTYPE	VARCHAR(128)	Name of the source data type of this data type.
		Nullable	Contains the null value if this is a built-in data type.
SYSTEM_TYPE_SCHEMA	SYSTSCHEMA	CHAR(10)	System schema name of the data type.
SYSTEM_TYPE_NAME	SYSTNAME	CHAR(10)	System name of the data type.
METATYPE	METATYPE	CHAR(1)	Indicates the type of data type. **A** Array data type **S** System predefined data type. **T** User-defined distinct type.

Table 197. SYSTYPES table (continued)

Column Name	System Column Name	Data Type	Description
LENGTH	LENGTH	INTEGER	The length attribute of the data type; or, in the case of a decimal, numeric, or nonzero precision binary column, its precision. For an array data type, it is the length of a single array element: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **Precision of number** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHARACTER **Maximum length of string** VARCHAR or CLOB **Length of graphic string** GRAPHIC **Maximum length of graphic string** VARGRAPHIC or DBCLOB **Length of binary string** BINARY **Maximum length of binary string** VARBINARY or BLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **Maximum length of datalink URL and comment** DATALINK **40 bytes** ROWID **2147483647 bytes** XML **Same value as the source type** DISTINCT
NUMERIC_SCALE	SCALE	SMALLINT	Scale of numeric data.
		Nullable	Contains the null value if the data type is not decimal, numeric, or binary.
CCSID	CCSID	INTEGER	The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB, XML, and DATALINK data types.
		Nullable	Contains the null value if the data type is numeric.

Table 197. SYSTYPES table (continued)

Column Name	System Column Name	Data Type	Description
STORAGE	STORAGE	INTEGER	The storage requirements for the data type: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **(Precision/2) + 1** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHAR **Maximum length of string + 2** VARCHAR **Maximum length of string + 29** CLOB **Length of string * 2** GRAPHIC **Maximum length of string * 2 + 2** VARGRAPHIC **Maximum length of string * 2 + 29** DBCLOB **Length of binary string** BINARY **Maximum length of binary string + 2** VARBINARY **Maximum length of string + 29** BLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **Maximum length of datalink URL and comment + 24** DATALINK **42 bytes** ROWID **2147483647 bytes + 29** XML **Same value as the source type** DISTINCT **Note:** This column supplies the storage requirements for all data types.
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	The precision of all numeric data types. **Note:** This column supplies the precision of all numeric data types, including decimal floating-point and single-and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the data type is not numeric.

Table 197. SYSTYPES table (continued)

Column Name	System Column Name	Data Type	Description
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER	Maximum length of the string for binary, character, and graphic string and XML data types.
		Nullable	Contains the null value if the data type is not a string.
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER	Number of bytes for binary, character, and graphic string and XML data types.
		Nullable	Contains the null value if the data type is not a string.
ALLOCATE	ALLOCATE	INTEGER	Allocated length of the string for binary, varying-length character, varying-length graphic, and XML data types.
		Nullable	Contains the null value if the data type is numeric or fixed-length, or an array data type.
NUMERIC_PRECISION_RADIX	RADIX	INTEGER	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits:
		Nullable	**2** Binary; floating-point precision is specified in binary digits.
			10 Decimal; all other numeric types are specified in decimal digits.
			Contains the null value if the data type is not numeric.
DATETIME_PRECISION	DATPRC	INTEGER	The fractional part of a date, time, or timestamp.
		Nullable	**0** For DATE and TIME data types
			6 For TIMESTAMP data types (number of microseconds).
			Contains the null value if the data type is not date, time, or timestamp.
CREATE_TIME	CRTTIME	TIMESTAMP	Identifies the timestamp when the data type was created.
		Nullable	
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement.
		Nullable	Contains the null value if there is no long comment.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number of the data type.
LAST_ ALTERED	ALTEREDTS	TIMESTAMP	Reserved. Contains the null value.
		Nullable	
NORMALIZE_DATA	NORMALIZE	VARCHAR(3)	Indicates whether the parameter value should be normalized or not. This attribute only applies to UTF-8 and UTF–16 data.
		Nullable	**NO** The value should not be normalized.
			YES The value should be normalized.
TYPE_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200	A character string supplied with the LABEL statement (type text).
		Nullable	Contains the null value if the type has no text.
MAXIMUM_CARDINALITY	MAXCARD	BIGINT	The maximum cardinality of the array data type.
		Nullable	Contains the null value if the type is not an array type.

SYSUDTAUTH

The SYSUDTAUTH view contains one row for every privilege granted on a type. Note that this catalog view cannot be used to determine whether a user is authorized to a type because the privilege to use a type could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the SYSUDTAUTH view:

Table 198. SYSUDTAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
USER_DEFINED_TYPE_SCHEMA	UDT_SCHEMA	VARCHAR(128)	Name of the schema
USER_DEFINED_TYPE_NAME	UDT_NAME	VARCHAR(128)	Name of the type
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the type. **USAGE** The privilege to use the type.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	If the privilege is granted through an authorization, contains the name of the authorization list. Contains the null value if the privilege is not granted through an authorization list.
SYSTEM_TYPE_SCHEMA	SYSTSCHEMA	CHAR(10)	System name of the schema
SYSTEM_TYPE_NAME	SYSTNAME	CHAR(10)	System name of the type

SYSVARIABLEAUTH

The SYSVARIABLEAUTH view contains one row for every privilege granted on a global variable. Note that this catalog view cannot be used to determine whether a user is authorized to a global variable because the privilege to use a global variable could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the SYSVARIABLEAUTH view:

Table 199. SYSVARIABLEAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
VARIABLE_SCHEMA	VARSCHEMA	VARCHAR(128)	Name of the schema
VARIABLE_NAME	VARNAME	VARCHAR(128)	Name of the global variable
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the global variable. **READ** The privilege to read the value of the global variable. **WRITE** The privilege to assign a value to the global variable.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	If the privilege is granted through an authorization, contains the name of the authorization list. Contains the null value if the privilege is not granted through an authorization list.
SYSTEM_VAR_SCHEMA	SYSVSCHEMA	CHAR(10)	System name of the schema
SYSTEM_VAR_NAME	SYSVNAME	CHAR(10)	System name of the global variable

SYSVARIABLEDEP

The SYSVARIABLEDEP table records the dependencies of variables.

The following table describes the columns in the SYSVARIABLEDEP table:

Table 200. SYSVARIABLEDEP table

Column name	System Column Name	Data Type	Description
VARIABLE_SCHEMA	VARSCHEMA	VARCHAR(128)	Schema name of the variable.
VARIABLE_NAME	VARNAME	VARCHAR(128)	Name of the variable.
OBJECT_SCHEMA	BSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the object.
OBJECT_NAME	BNAME	VARCHAR(128)	Name of the object the variable is dependent on.
OBJECT_TYPE	BTYPE	CHAR(24)	Indicates the object type of the object referenced in the variable: **ALIAS** The object is an alias. **FUNCTION** The object is a function. **MATERIALIZED QUERY TABLE** The object is a materialized query table. **SCHEMA** The object is a schema. **SEQUENCE** The object is a sequence. **TABLE** The object is a table. **TYPE** The object is a distinct type. **VARIABLE** The object is a variable. **VIEW** The object is a view.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number of the object.
PARM_SIGNATURE	SIGNATURE	VARCHAR(10000)	This column identifies the routine signature.
		Nullable	Contains the null value if the object is not a routine.

SYSVARIABLES

The SYSVARIABLES table contains one row for each global variable.

The following table describes the columns in the SYSVARIABLES table:

Table 201. SYSVARIABLES table

Column Name	System Column Name	Data Type	Description
VARIABLE_SCHEMA	VARSCHEMA	VARCHAR(128)	Schema name of the variable.
VARIABLE_NAME	VARNAME	VARCHAR(128)	Name of the variable.
SYSTEM_VAR_SCHEMA	SYSVSCHEMA	CHAR(10)	System schema name.
SYSTEM_VAR_NAME	SYSVNAME	CHAR(10)	System variable name.
VARIABLE_OWNER	OWNER	VARCHAR(128)	Authorization ID of the owner of the global variable.
VARIABLE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that created the variable.
VARIABLE_CREATED	CREATDTS	TIMESTAMP	Identifies the timestamp when the variable was created.
SCOPE	SCOPE	CHAR(1)	Indicates the scope of the variable:. **S** Session
OWNER_TYPE	OWNER_TYPE	CHAR(1)	Indicates the owner of the variable: **U** The owner is an idividual user
USER_DEFINED_TYPE_SCHEMA	TYPESCHEMA	VARCHAR(128)	Schema of the data type if this is a distinct type.
		Nullable	Contains the null value if the variable is not a distinct type.
USER_DEFINED_TYPE_NAME	TYPENAME	VARCHAR(128)	Name of the data type if this is a distinct type.
		Nullable	Contains the null value if the variable is not a distinct type.

Table 201. SYSVARIABLES table (continued)

Column Name	System Column Name	Data Type	Description
DATA_TYPE	DATA_TYPE	VARCHAR(128)	Type of variable: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **FLOAT** Floating point; FLOAT, REAL, or DOUBLE PRECISION **DECFLOAT** Decimal floating-point **CHAR** Fixed-length character string **VARCHAR** Varying-length character string **CLOB** Character large object string **GRAPHIC** Fixed-length graphic string **VARG** Varying-length graphic string **DBCLOB** Double-byte character large object string **BINARY** Fixed-length binary string **VARBIN** Varying-length binary string **BLOB** Binary large object string **DATE** Date **TIME** Time **TIMESTMP** Timestamp **XML** XML **DISTINCT** Distinct type

Table 201. SYSVARIABLES table (continued)

Column Name	System Column Name	Data Type	Description
LENGTH	LENGTH	INTEGER	The length attribute of the data type; or, in the case of a decimal, numeric, or nonzero precision binary variable, its precision: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **Precision of number** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHARACTER **Maximum length of string** VARCHAR or CLOB **Length of graphic string** GRAPHIC **Maximum length of graphic string** VARGRAPHIC or DBCLOB **Length of binary string** BINARY **Maximum length of binary string** VARBINARY or BLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **2147483647 bytes** XML **Same value as the source type** DISTINCT
NUMERIC_SCALE	SCALE	INTEGER	Scale of numeric data.
		Nullable	Contains the null value if the data type is not decimal, numeric, or binary.

Table 201. SYSVARIABLES table (continued)

Column Name	System Column Name	Data Type	Description
STORAGE	STORAGE	INTEGER	The storage requirements for the variable: **8 bytes** BIGINT **4 bytes** INTEGER **2 bytes** SMALLINT **(Precision/2) + 1** DECIMAL **Precision of number** NUMERIC **8 bytes** FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION **4 bytes** FLOAT(n) where n = 1 to 24, or REAL **8 bytes** DECFLOAT(16) **16 bytes** DECFLOAT(34) **Length of string** CHAR **Maximum length of string + 2** VARCHAR **Maximum length of string + 29** CLOB **Length of string * 2** GRAPHIC **Maximum length of string * 2 + 2** VARGRAPHIC **Maximum length of string * 2 + 29** DBCLOB **Length of binary string** BINARY **Maximum length of binary string + 2** VARBINARY **Maximum length of string + 29** BLOB **4 bytes** DATE **3 bytes** TIME **10 bytes** TIMESTAMP **2147483647 +29 bytes** XML **Same value as the source type** DISTINCT **Note:** This column supplies the storage requirements for all data types.
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	The precision of all numeric data types. **Note:** This column supplies the precision of all numeric data types, including decimal floating-point and single-and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the data type is not numeric.

Table 201. SYSVARIABLES table (continued)

Column Name	System Column Name	Data Type	Description
CCSID	CCSID	INTEGER Nullable	The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB, XML, and DATALINK data types. Contains the null value if the data type is numeric.
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER Nullable	Maximum length of the string for binary, character, and graphic string data types and the XML data type. Contains the null value if the data type is not a string or XML.
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER Nullable	Number of bytes for binary, character, and graphic string data types and the XML data type. Contains the null value if the data type is not a string or XML.
NUMERIC_PRECISION_RADIX	RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the data type is not numeric.
DATETIME_PRECISION	DATPRC	INTEGER Nullable	The fractional part of a date, time, or timestamp. **0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if the data type is not date, time, or timestamp.
DEFAULT	DEFAULT	DBCLOB(2M) CCSID 1200	The expression used to calculate the initial value of the global variable when it is first referenced.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number of the variable.
VARIABLE_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	A character string supplied with the LABEL statement (type text). Contains the null value if the type has no text.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
DEFAULT_SCHEMA	QUALIFIER	VARCHAR(128) Nullable	Qualifier for unqualifed tables and views.
SQL_PATH	SQL_PATH	VARCHAR(3483) Nullable	Identifies the path.

Table 201. SYSVARIABLES table (continued)

Column Name	System Column Name	Data Type	Description	
ROUNDING_MODE	DECFLTRND	CHAR(1)	Identifies the DECFLOAT rounding mode.	
			C	ROUND_CEILING
			D	ROUND_DOWN
			F	ROUND_FLOOR
			G	ROUND_HALF_DOWN
			E	ROUND_HALF_EVEN
			H	ROUND_HALF_UP
			U	ROUND_UP

SYSVIEWDEP

The SYSVIEWDEP view records the dependencies of views on tables, including the views of the SQL catalog.

The following table describes the columns in the SYSVIEWDEP view:

Table 202. SYSVIEWDEP view

Column name	System Column Name	Data Type	Description
VIEW_NAME	DNAME	VARCHAR(128)	Name of the view. This is the SQL view name if it exists; otherwise, it is the system view name.
VIEW_OWNER	DCREATOR	VARCHAR(128)	Owner of the view
OBJECT_NAME	ONAME	VARCHAR(128)	Name of the object the view is dependent on.
OBJECT_SCHEMA	OSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the object the view is dependent on.
OBJECT_TYPE	OTYPE	CHAR(24)	Type of object the view was based on: **FUNCTION** Function **MATERIALIZED QUERY TABLE** The object is a materialized query table. **TABLE** Table **TYPE** Distinct type **VARIABLE** The object is a variable. **VIEW** View
VIEW_SCHEMA	DDBNAME	VARCHAR(128)	Name of the schema of the view.
SYSTEM_VIEW_NAME	SYS_VNAME	CHAR(10)	System View name
SYSTEM_VIEW_SCHEMA	SYS_VDNAME	CHAR(10)	System View schema
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10) Nullable	System Table name. Contains the null value if the object is a function or distinct type.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10) Nullable	System Table schema. Contains the null value if the object is a function or distinct type.
TABLE_NAME	BNAME	VARCHAR(128) Nullable	Name of the table or view the view is dependent on. This is the SQL view name if it exists; otherwise, it is the system view name. Contains the null value if the object is a function or distinct type.
TABLE_OWNER	BCREATOR	VARCHAR(128) Nullable	Owner of the table or view the view is dependent on. Contains the null value if the object is a function or distinct type.
TABLE_SCHEMA	BDBNAME	VARCHAR(128) Nullable	Name of the SQL schema that contains the table or view the view is dependent on. Contains the null value if the object is a function or distinct type.

Table 202. SYSVIEWDEP view (continued)

Column name	System Column Name	Data Type	Description
TABLE_TYPE	BTYPE	CHAR(1)	Type of object the view was based on:
		Nullable	**T** Table **P** Physical file **M** Materialized query table **V** View **L** Logical file Contains the null value if the object is a function or distinct type.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
PARM_SIGNATURE	SIGNATURE	VARCHAR(10000)	This column identifies the routine signature.
		Nullable	Contains the null value if the object is not a routine.

SYSVIEWS

The SYSVIEWS view contains one row for each view in the SQL schema, including the views of the SQL catalog.

The following table describes the columns in the SYSVIEWS view:

Table 203. SYSVIEWS view

Column Name	System Column Name	Data Type	Description
TABLE_NAME	NAME	VARCHAR(128)	Name of the view. This is the SQL view name if it exists; otherwise, it is the system view name.
VIEW_OWNER	CREATOR	VARCHAR(128)	Owner of the view
SEQNO	SEQNO	INTEGER	Sequence number of this row; will always be 1.
CHECK_OPTION	CHECK	CHAR(1)	The check option used on the view **N** No check option was specified **Y** The local option was specified **C** The cascaded option was specified
VIEW_DEFINITION	TEXT	VARGRAPHIC(5000) CCSID 1200 Nullable	The query expression portion of the CREATE VIEW statement. Contains the null value if the view definition cannot be contained in the column without truncation.
IS_UPDATABLE	UPDATES	CHAR(1)	Specifies if the view is updatable: **Y** The view is updatable **N** The view is not updatable
TABLE_SCHEMA	DBNAME	VARCHAR(128)	Name of the SQL schema that contains the view.
SYSTEM_VIEW_NAME	SYS_VNAME	CHAR(10)	System view name
SYSTEM_VIEW_SCHEMA	SYS_VDNAME	CHAR(10)	System view schema name
IS_INSERTABLE_INTO	INSERTABLE	VARCHAR(3)	Identifies whether an INSERT is allowed on the view. **NO** An INSERT is not allowed on this view. **YES** An INSERT is allowed on this view.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
IS_DELETABLE	DELETES	CHAR(1) Nullable	Specifies if the view is deletable: **Y** The view is deletable **N** The view is read-only
VIEW_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the view.
ROUNDING_MODE	DECFLTRND	CHAR(1) Nullable	Indicates the DECFLOAT rounding mode of the view: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP Contains the null value if the view does not have an expression that references a DECFLOAT column, function, or constant.

SYSXSROBJECTAUTH

The SYSXSROBJECTAUTH view contains one row for every privilege granted on an XML schema. Note that this catalog view cannot be used to determine whether a user is authorized to a XML schema because the privilege to use a XML schema could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the SYSXSROBJECTAUTH view:

Table 204. SYSXSROBJECTAUTH view

Column Name	System Column Name	Data Type	Description
GRANTOR	GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
XSROBJECTSCHEMA	XSRSCHEMA	VARCHAR(128)	Name of the schema
XSROBJECTNAME	XSRNAME	VARCHAR(128)	Name of the XML schema
PRIVILEGE_TYPE	PRIVTYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the XML schema. **USAGE** The privilege to use the XML schema.
IS_GRANTABLE	GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
AUTHORIZATION_LIST	AUTL	VARCHAR(10) Nullable	If the privilege is granted through an authorization, contains the name of the authorization list. Contains the null value if the privilege is not granted through an authorization list.
SYSTEM_XSR_SCHEMA	SYSXSCHEMA	CHAR(10)	System name of the schema
SYSTEM_XSR_NAME	SYSXNAME	CHAR(10)	System name of the XML schema

XSRANNOTATIONINFO

The XSRANNOTATIONINFO table contains one row for each annotation in an XML schema to record the table and column information about the annotation.

The following table describes the columns in the XSRANNOTATIONINFO table:

Table 205. XSRANNOTATIONINFO table

Column name	System Column Name	Data Type	Description
XSROBJECTID	OBJECTID	BIGINT	Internal identifier of the XML schema.
ANNOTATION_ID	ANNOID	INTEGER	Internal identifier of the XML schema annotation.
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Schema of the table to insert the data into.
TABLE_NAME	TABNAME	VARCHAR(128)	Table to insert the data into.
ROWSET	ROWSET	VARCHAR(1000)	Name of the rowset for this annotation.
COLUMN_NAME	COLNAME	VARCHAR(128)	Name of the column for this annotation.
DATA_TYPE	COLTYPE	INTEGER	Data type of the column for this annotation.
INSTANCE_TYPE	INSTTYPE	INTEGER	Type of data that will be provided by parser during decomposition: **2** decimal **4** long integer **5** integer **6** short integer **16** string **30** datetime **41** float **42** double
TRUNCATE	TRUNCATE	INTEGER	Indication of whether data can be truncated: **0** Data cannot be truncated **1** Data can be truncated
EXPRESSION	EXPRESSION	VARCHAR(1024) Nullable	Expression to be applied to data on insert by DB2.
CONDITION	CONDITION	VARCHAR(1024) Nullable	Condition to be applied before data is inserted by DB2.
CAST_EXPRESSION	CASTEXP	VARCHAR(20) Nullable	Cast expression to be applied when the column data is inserted by DB2 during decomposition.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
ROWSET_ORDER	ORDER	INTEGER	RowSet order sequence. Default is 1.

XSROBJECTCOMPONENTS

The XSROBJECTCOMPONENTS table contains one row for each component (document) in an XML schema.

The following table describes the columns in the XSROBJECTCOMPONENTS table:

Table 206. XSROBJECTCOMPONENTS table

Column name	System Column Name	Data Type	Description
XSRCOMPONENTID	COMPID	BIGINT	Internal identifier of the XML schema document.
TARGETNAMESPACE	TGTNAMESPC	VARCHAR(1000) Nullable	String identifier for the target namespace.
SCHEMALOCATION	SCHEMALOC	VARCHAR(1000) Nullable	String identifier for the schema location.
COMPONENT	COMPONENT	BLOB(30M) Nullable	Contents of the XML schema document.
PROPERTIES	PROPERTIES	BLOB(5M) Nullable	Additional document property information for the XML schema document.
CREATED	CREATEDTS	TIMESTAMP	The time when the XSR_REGISTER stored procedure was executed for the XML schema.
STATUS	STATUS	CHAR(1)	Registration status of the XML schema: **C** Complete **I** Incomplete
IASP_NUMBER	IASPNUMBER	SMALLINT	Indicates the IASP where the catalog table is located.

XSROBJECTHIERARCHIES

The XSROBJECTHIERARCHIES table contains one row for each component (document) in an XML schema to record the XML schema document hierarchy relationship.

The following table describes the columns in the XSROBJECTHIERARCHIES table:

Table 207. XSROBJECTHIERARCHIES table

Column name	System Column Name	Data Type	Description
XSROBJECTID	OBJECTID	BIGINT	Internal identifier of the XML schema.
XSRCOMPONENTID	COMPID	BIGINT	Internal identifier of the XML schema document.
HTYPE	HTYPE	CHAR(1)	Hierarchy type: **D** Document **P** Primary document
TARGETNAMESPACE	TGTNAMESPC	VARCHAR(1000) Nullable	String identifier for the target namespace.
SCHEMALOCATION	SCHEMALOC	VARCHAR(1000) Nullable	String identifier for the schema location.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

XSROBJECTS

The XSROBJECTS table contains one row for each registered XML schema.

The following table describes the columns in the XSROBJECTS table:

Table 208. XSROBJECTS table

Column name	System Column Name	Data Type	Description
XSROBJECTID	OBJECTID	BIGINT	Internal identifier of the XML schema.
XSROBJECTSCHEMA	XSRSCHEMA	VARCHAR(128)	Qualifier of the XML schema name.
XSROBJECTNAME	XSRNAME	VARCHAR(128)	Name of the XML schema
SYSTEM_XSR_SCHEMA	SYSXSCHEMA	CHAR(10)	System library name corresponding to the qualifier of the XML schema.
SYSTEM_XSR_NAME	SYSXNAME	CHAR(10)	System name of the XSROBJECT that correlates to the XML schema.
TARGETNAMESPACE	TGTNAMESPC	VARCHAR(1000) Nullable	String identifier for the target namespace.
SCHEMALOCATION	SCHEMALOC	VARCHAR(1000) Nullable	String identifier for the schema location.
XSR_OBJECT_TEXT	LABEL	VARGRAPHIC(50) CCSID 1200 Nullable	A character string supplied with the LABEL statement. Contains the null value if there is no label.
GRAMMAR	GRAMMAR	BLOB(250M) Nullable	The internal binary representation of the XML schema as the result of an XML_COMPLETE.
PROPERTIES	PROPERTIES	BLOB(5M) Nullable	Additional document property information for the XML schema document.
XSR_SCHEMA_DEFINER	DEFINER	VARCHAR(128)	Authorization ID under which the XML schema was created.
XSR_SCHEMA_OWNER	OWNER	VARCHAR(128)	Authorization ID that owns the XML schema.
XSR_SCHEMA_CREATED	CREATEDTS	TIMESTAMP	The time when the XML schema document was registered.
STATUS	STATUS	CHAR(1)	Registration status of the XML schema: **C** Complete **I** Incomplete
DECOMPOSITION	DECOMP	CHAR(1) Nullable	Indicates the decomposition status of the XSR object: **Y** Enabled **N** Not enabled
VERSION	VERSION	VARCHAR(128) Nullable	Indicates the version used during decomposition.
IASP_NUMBER	IASPNUMBER	SMALLINT	Indicates the IASP where the catalog table is located.
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.

ODBC and JDBC catalog views

The catalog includes the views and tables in the SYSIBM library displayed in this section.

View Name	Description
"SQLCOLPRIVILEGES" on page 1723	Information about privileges granted on columns
"SQLCOLUMNS" on page 1724	Information about column attributes
"SQLFOREIGNKEYS" on page 1730	Information about foreign keys
"SQLFUNCTIONCOLS" on page 1731	Information about function parameters
"SQLFUNCTIONS" on page 1737	Information about functions
"SQLPRIMARYKEYS" on page 1738	Information about primary keys
"SQLPROCEDURECOLS" on page 1739	Information about procedure parameters
"SQLPROCEDURES" on page 1745	Information about procedures
"SQLSCHEMAS" on page 1746	Information about schemas
"SQLSPECIALCOLUMNS" on page 1747	Information about columns of a table that can be used to uniquely identify a row
"SQLSTATISTICS" on page 1751	Statistical information about tables
"SQLTABLEPRIVILEGES" on page 1752	Information about privileges granted on tables
"SQLTABLES" on page 1753	Information about tables
"SQLTYPEINFO" on page 1754	Information about the types of tables
"SQLUDTS" on page 1760	Information about built-in data types and distinct types

SQLCOLPRIVILEGES

The SQLCOLPRIVILEGES view contains one row for every privilege granted on a column or a privilege granted on the column's table. Note that this catalog view cannot be used to determine whether a user is authorized to a column because the privilege to use a column could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the view:

Table 209. SQLCOLPRIVILEGES view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name.
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	VARCHAR(128)	Table name.
COLUMN_NAME	VARCHAR(128)	Column name.
GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
PRIVILEGE	VARCHAR(10)	The privilege granted: **UPDATE** The privilege to update the column. **REFERENCES** The privilege to reference the column in a referential constraint.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
DBNAME	VARCHAR(8) Nullable	Reserved. The column contains the null value.

SQLCOLUMNS

The SQLCOLUMNS view contains one row for every column in a table, view, or alias.

The following table describes the columns in the view:

Table 210. SQLCOLUMNS view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name.
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	VARCHAR(128)	Table name.
COLUMN_NAME	VARCHAR(128)	Column name.
DATA_TYPE	SMALLINT	The data type of the column: **–5** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **-99** CLOB **–95** GRAPHIC **–96** VARGRAPHIC **–350** DBCLOB **–8** NCHAR **–9** NVARCHAR **–10** NCLOB **-98** BLOB **91** DATE **92** TIME **93** TIMESTAMP **70** DATALINK **–100** ROWID **-370** XML **17** DISTINCT

Table 210. SQLCOLUMNS view (continued)

Column Name	Data Type	Description
TYPE_NAME	VARCHAR(261)	The name of the data type of the column: **BIGINT** BIGINT **INTeger** INTEGER **SMALLINT** SMALLINT **DECIMAL** DECIMAL **NUMERIC** NUMERIC **FLOAT** DOUBLE PRECISION **REAL** REAL **DECFLOAT** DECFLOAT **CHARacter** CHARACTER **CHARacter FOR BIT DATA** CHARACTER FOR BIT DATA **VARCHAR** VARCHAR **VARCHAR FOR BIT DATA** VARCHAR FOR BIT DATA **CLOB** CLOB **GRAPHIC** GRAPHIC **VARGRAPHIC** VARGRAPHIC **DBCLOB** DBCLOB **NCHAR** NCHAR **NVARCHAR** NVARCHAR **NCLOB** NCLOB **BINARY** BINARY **VARBINARY** VARBINARY **BLOB** BLOB **DATE** DATE **TIME** TIME **TIMESTAMP** TIMESTAMP **DATALINK** DATALINK **ROWID** ROWID **XML** XML **Qualified Type Name** DISTINCT
COLUMN_SIZE	INTEGER	The length of the column.
BUFFER_LENGTH	INTEGER	Indicates the length of the column in a buffer.
DECIMAL_DIGITS	SMALLINT	Indicates the number of digits for a numeric column.
	Nullable	Contains the null value if the object is not numeric or timestamp.

Table 210. SQLCOLUMNS view (continued)

Column Name	Data Type	Description
NUM_PREC_RADIX	SMALLINT	Indicates the radix of a numeric column.
	Nullable	Contains the null value if the object is not numeric.
NULLABLE	SMALLINT	Indicates whether the column can contain the null value. **0** The column does not allow nulls. **1** The column does allow nulls.
REMARKS	VARCHAR(2000)	A character string supplied with the COMMENT statement.
	Nullable	Contains the null value if there is no long comment.
COLUMN_DEF	VARCHAR(2000)	The default value of the column.
	Nullable	Contains the null value if there is no default value.
SQL_DATA_TYPE	SMALLINT	Indicates the SQL data type of the column. **–5** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **-99** CLOB **–95** GRAPHIC **–96** VARGRAPHIC **–350** DBCLOB **–8** NCHAR **–9** NVARCHAR **–10** NCLOB **-98** BLOB **9** DATE **9** TIME **9** TIMESTAMP **70** DATALINK **–100** ROWID **-370** XML **17** DISTINCT
SQL_DATETIME_SUB	SMALLINT Nullable	The datetime subtype of the data type: **1** DATE **2** TIME **3** TIMESTAMP Contains the null value if the column is not a datetime data type.

Table 210. SQLCOLUMNS view (continued)

Column Name	Data Type	Description
CHAR_OCTET_LENGTH	INTEGER	Indicates the length in bytes of the column.
	Nullable	Contains the null value if the column is not a string.
ORDINAL_POSITION	INTEGER	Indicates the ordinal position of the column in the table.
IS_NULLABLE	VARCHAR(3)	Indicates whether the column can contain the null value.
		NO The column is not nullable.
		YES The column is nullable.
JDBC_DATA_TYPE	SMALLINT	Indicates the JDBC data type of the column.
		–5 BIGINT
		4 INTEGER
		5 SMALLINT
		3 DECIMAL
		2 NUMERIC
		8 DOUBLE PRECISION
		7 REAL
		1111 DECFLOAT
		1 CHARACTER or GRAPHIC
		-15 NCHAR
		–2 CHARACTER FOR BIT DATA or BINARY
		12 VARCHAR or VARGRAPHIC
		-9 NVARCHAR
		–3 VARCHAR FOR BIT DATA or VARBINARY
		2005 CLOB or DBCLOB
		2011 NCLOB
		2004 BLOB
		91 DATE
		92 TIME
		93 TIMESTAMP
		70 DATALINK
		-8 ROWID
		2009 XML
		2001 DISTINCT
SCOPE_CATALOG	VARCHAR(128)	Reserved. Contains the null value.
	Nullable	
SCOPE_SCHEMA	VARCHAR(128)	Reserved. Contains the null value.
	Nullable	
SCOPE_TABLE	VARCHAR(128)	Reserved. Contains the null value.
	Nullable	
SOURCE_DATA_TYPE	SMALLINT	The source data type if the data type of the column is a distinct type. For values see JDBC_DATA_TYPE.
	Nullable	Contains the null value if the data type is not a distinct type.
DBNAME	VARCHAR(8)	Reserved. Contains the null value.
	Nullable	

Table 210. SQLCOLUMNS view (continued)

Column Name	Data Type	Description
PSEUDO_COLUMN	SMALLINT	Indicates whether this is a ROWID, identity, or row change timestamp column. **1** The column is not a ROWID, identity, or row change timestamp column. **2** The column is a ROWID, identity, or row change timestamp column.
COLUMN_TEXT	VARCHAR(50)	The text of the column.
	Nullable	Contains the null value if the column has no column text.
SYSTEM_COLUMN_NAME	CHAR(10)	The system name of the column.
I_DATA_TYPE	SMALLINT	Indicates the IBM i CLI data type of the column. **19** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **14** CLOB **95** GRAPHIC or NCHAR **96** VARGRAPHIC or NVARCHAR **15** DBCLOB or NCLOB **13** BLOB **91** DATE **92** TIME **93** TIMESTAMP **16** DATALINK **1111** ROWID **-370** XML **2001** DISTINCT
HIDDEN	CHAR(1)	Specifies whether the column is included in an implicit column list. **P** Partially hidden. **N** Not hidden.

Table 210. SQLCOLUMNS view (continued)

Column Name	Data Type	Description
HAS_DEFAULT	CHAR(1)	If the column has a default value (DEFAULT clause or null capable): **N** No **Y** Yes **A** The column has a ROWID data type and the GENERATED ALWAYS attribute. **D** The column has a ROWID data type and the GENERATED BY DEFAULT attribute. **E** The column is defined with the FOR EACH ROW ON UPDATE and the GENERATED ALWAYS attribute. **F** The column is defined with the FOR EACH ROW ON UPDATE and the GENERATED BY DEFAULT attribute. **I** The column is defined with the AS IDENTITY and GENERATED ALWAYS attributes. **J** The column is defined with the AS IDENTITY and GENERATED BY DEFAULT attributes. If the column is for a view, N is returned.
SOURCE_TYPE_NAME	VARCHAR(128) Nullable	If the column data type is a user-defined type, the built-in data type name of its source type. Contains the null value if the column data type is not a user-defined type.
SOURCE_SQL_DATA_TYPE	SMALLINT Nullable	If the column data type is a user-defined type, the built-in SQL_DATA_TYPE of its source type. For values see SQL_DATA_TYPE. Contains the null value if the column data type is not a user-defined type.
SOURCE_JDBC_DATA_TYPE	SMALLINT Nullable	If the column data type is a user-defined type, the built-in JDBC_DATA_TYPE of its source type. For values see JDBC_DATA_TYPE. Contains the null value if the column data type is not a user-defined type.

SQLFOREIGNKEYS

The SQLFOREIGNKEYS view contains one row for every referential constraint key on a table.

The following table describes the columns in the view:

Table 211. SQLFOREIGNKEYS view

Column Name	Data Type	Description
PKTABLE_CAT	VARCHAR(128)	Relational database name
PKTABLE_SCHEM	VARCHAR(128)	Name of the SQL schema containing the parent table.
PKTABLE_NAME	VARCHAR(128)	Parent table name.
PKCOLUMN_NAME	VARCHAR(128)	Parent key column name.
FKTABLE_CAT	VARCHAR(128)	Relational database name
FKTABLE_SCHEM	VARCHAR(128)	Name of the SQL schema containing the dependent table of the referential constraint.
FKTABLE_NAME	VARCHAR(128)	Dependent table name of the referential constraint.
FKCOLUMN_NAME	VARCHAR(128)	Dependent key name.
KEY_SEQ	SMALLINT	The position of the column within the key.
UPDATE_RULE	SMALLINT	Update Rule. **1** RESTRICT **3** NO ACTION
DELETE_RULE	SMALLINT	Delete Rule: **0** CASCADE **1** RESTRICT **2** SET NULL **3** NO ACTION **4** SET DEFAULT
FK_NAME	VARCHAR(128)	Name of the referential constraint
PK_NAME	VARCHAR(128)	Name of the unique constraint
DEFERRABILITY	SMALLINT	Indicates whether the constraint checking can be deferred. Will always be 7.
UNIQUE_OR_PRIMARY	CHAR(7)	Indicates the type of parent constraint: **PRIMARY** The parent constraint is a primary key. **UNIQUE** The parent constraint is a unique constraint.

SQLFUNCTIONCOLS

The SQLFUNCTIONCOLS view contains one row for every parameter of a function. The result of a scalar function and the result columns of a table function are also returned.

The following table describes the columns in the view:

Table 212. SQLFUNCTIONCOLS view

Column Name	Data Type	Description
FUNCTION_CAT	VARCHAR(128)	Relational database name
FUNCTION_SCHEM	VARCHAR(128)	Schema name of the function instance.
FUNCTION_NAME	VARCHAR(128)	Name of the function instance.
COLUMN_NAME	VARCHAR(128)	Name of a function parameter.
	Nullable	Contains the null value if the parameter does not have a name.
COLUMN_TYPE	SMALLINT	Type of the parameter:
		1 IN
		4 RETURN value
		5 Result column of a table function
DATA_TYPE	SMALLINT	The data type of the parameter:
		–5 BIGINT
		4 INTEGER
		5 SMALLINT
		3 DECIMAL
		2 NUMERIC
		8 DOUBLE PRECISION
		7 REAL
		-360 DECFLOAT
		1 CHARACTER
		–2 CHARACTER FOR BIT DATA or BINARY
		12 VARCHAR
		–3 VARCHAR FOR BIT DATA or VARBINARY
		-99 CLOB
		–95 GRAPHIC or
		–96 VARGRAPHIC
		–350 DBCLOB
		–8 NCHAR
		–9 NVARCHAR
		–10 NCLOB
		-98 BLOB
		91 DATE
		92 TIME
		93 TIMESTAMP
		70 DATALINK
		–100 ROWID
		-370 XML
		17 DISTINCT
		50 ARRAY

Table 212. SQLFUNCTIONCOLS view (continued)

Column Name	Data Type	Description
TYPE_NAME	VARCHAR(261)	The name of the data type of the parameter: **BIGINT** BIGINT **INTeger** INTEGER **SMALLINT** SMALLINT **DECIMAL** DECIMAL **NUMERIC** NUMERIC **FLOAT** DOUBLE PRECISION **REAL** REAL **DECFLOAT** DECFLOAT **CHARacter** CHARACTER **CHARacter FOR BIT DATA** CHARACTER FOR BIT DATA **VARCHAR** VARCHAR **VARCHAR FOR BIT DATA** VARCHAR FOR BIT DATA **CLOB** CLOB **GRAPHIC** GRAPHIC **VARGRAPHIC** VARGRAPHIC **DBCLOB** DBCLOB **NCHAR** NCHAR **NVARCHAR** NVARCHAR **NCLOB** NCLOB **BINARY** BINARY **VARBINARY** VARBINARY **BLOB** BLOB **DATE** DATE **TIME** TIME **TIMESTAMP** TIMESTAMP **DATALINK** DATALINK **ROWID** ROWID **XML** XML **Qualified Type Name** DISTINCT **Array Type Name** ARRAY
COLUMN_SIZE	INTEGER	Length of the parameter.
BUFFER_LENGTH	INTEGER	Indicates the length of the parameter in a buffer.

Table 212. SQLFUNCTIONCOLS view (continued)

Column Name	Data Type	Description
DECIMAL_DIGITS	SMALLINT	Scale of numeric or datetime data.
	Nullable	Contains the null value if the parameter is not decimal, numeric, binary, time or timestamp.
NUM_PREC_RADIX	SMALLINT	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits:
	Nullable	**2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the parameter is not numeric.
NULLABLE	SMALLINT	Indicates whether the parameter is nullable. **0** The parameter does not allow nulls. **1** The parameter does allow nulls.
REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
	Nullable	
COLUMN_DEF	DBCLOB(64K)	The expression string used to calculate the default value of a parameter, if one exists. If the default value is the null value, the expression string is the keyword NULL. Contains the null value if the parameter has no default.
	CCSID(1200) Nullable	

Table 212. SQLFUNCTIONCOLS view (continued)

Column Name	Data Type	Description
SQL_DATA_TYPE	SMALLINT	The SQL data type of the parameter: **–5** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **-99** CLOB **–95** GRAPHIC **–96** VARGRAPHIC **–350** DBCLOB **–8** NCHAR **–9** NVARCHAR **–10** NCLOB **-98** BLOB **9** DATE **9** TIME **9** TIMESTAMP **70** DATALINK **–100** ROWID **-370** XML **17** DISTINCT **50** ARRAY
SQL_DATETIME_SUB	SMALLINT Nullable	The datetime subtype of the parameter: **1** DATE **2** TIME **3** TIMESTAMP Contains the null value if the data type is not a datetime data type.
CHAR_OCTET_LENGTH	INTEGER Nullable	Indicates the length in characters of the parameter. Contains the null value if the column is not a string.
ORDINAL_POSITION	INTEGER	Numeric place of the parameter in the parameter list, ordered from left to right. For scalar functions, the result of the function has a value of 0. For table functions, the result columns are numbered from 1 (leftmost result column) to *n* (*n*th result column).

Table 212. SQLFUNCTIONCOLS view (continued)

Column Name	Data Type	Description	
IS_NULLABLE	VARCHAR(3)	Indicates whether the parameter is nullable.	
		NO	The parameter does not allow nulls.
		YES	The parameter does allow nulls.
SPECIFIC_NAME	VARCHAR(128)	Specific name of the function instance	
JDBC_DATA_TYPE	INTEGER	The JDBC data type of the parameter:	
		–5	BIGINT
		4	INTEGER
		5	SMALLINT
		3	DECIMAL
		2	NUMERIC
		8	DOUBLE PRECISION
		7	REAL
		1111	DECFLOAT
		1	CHARACTER or GRAPHIC
		-15	NCHAR
		–2	CHARACTER FOR BIT DATA or BINARY
		12	VARCHAR or VARGRAPHIC
		-9	NVARCHAR
		–3	VARCHAR FOR BIT DATA or VARBINARY
		2005	CLOB or DBCLOB
		2011	NCLOB
		2004	BLOB
		91	DATE
		92	TIME
		93	TIMESTAMP
		70	DATALINK
		-8	ROWID
		2009	XML
		2001	DISTINCT
		2003	ARRAY

Table 212. SQLFUNCTIONCOLS view (continued)

Column Name	Data Type	Description
I_DATA_TYPE	INTEGER	Indicates the IBM i CLI data type of the parameter. **19** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **14** CLOB **95** GRAPHIC or NCHAR **96** VARGRAPHIC or NVARCHAR **15** DBCLOB or NCLOB **13** BLOB **91** DATE **92** TIME **93** TIMESTAMP **16** DATALINK **1111** ROWID **-370** XML **2001** DISTINCT **50** ARRAY
SOURCE_DATA_TYPE	SMALLINT Nullable	The source data type if the data type of the parameter is a distinct type. For values see JDBC_DATA_TYPE. Contains the null value if the data type is not a distinct type.
SOURCE_TYPE_NAME	VARCHAR(128) Nullable	If the parameter data type is a user-defined type, the built-in data type name of its source type. Contains the null value if the parameter data type is not a user-defined type.
SOURCE_SQL_DATA_TYPE	SMALLINT Nullable	If the parameter data type is a user-defined type, the built-in SQL_DATA_TYPE of its source type. For values see SQL_DATA_TYPE. Contains the null value if the parameter data type is not a user-defined type.
SOURCE_JDBC_DATA_TYPE	SMALLINT Nullable	If the parameter data type is a user-defined type, the built-in JDBC_DATA_TYPE of its source type. For values see JDBC_DATA_TYPE. Contains the null value if the parameter data type is not a user-defined type.
MAXIMUM_CARDINALITY	BIGINT Nullable	The maximum cardinality of the array data type. Contains the null value if the type is not an array type.

SQLFUNCTIONS

The SQLFUNCTIONS view contains one row for every function.

The following table describes the columns in the view:

Table 213. SQLFUNCTIONS view

Column Name	Data Type	Description
FUNCTION_CAT	VARCHAR(128)	Relational database name
FUNCTION_SCHEM	VARCHAR(128)	Name of the schema of the function instance.
FUNCTION_NAME	VARCHAR(128)	Name of the function.
REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
FUNCTION_TYPE	SMALLINT	Type of function: **1** Column or scalar function **2** Table function
SPECIFIC_NAME	VARCHAR(128)	Specific name of the function instance

SQLPRIMARYKEYS

The SQLPRIMARYKEYS view contains one row for every primary constraint key on a table.

The following table describes the columns in the view:

Table 214. SQLPRIMARYKEYS view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the schema containing the table with the primary key.
TABLE_NAME	VARCHAR(128)	Name of the table with the primary key.
COLUMN_NAME	VARCHAR(128)	Name of a primary key column.
KEY_SEQ	SMALLINT	The position of the column within the key.
PK_NAME	VARCHAR(128)	Name of the primary key constraint.

SQLPROCEDURECOLS

The SQLPROCEDURECOLS view contains one row for every parameter of a procedure.

The following table describes the columns in the view:

Table 215. SQLPROCEDURECOLS view

Column Name	Data Type	Description
PROCEDURE_CAT	VARCHAR(128)	Relational database name
PROCEDURE_SCHEM	VARCHAR(128)	Schema name of the procedure instance.
PROCEDURE_NAME	VARCHAR(128)	Name of the procedure instance.
COLUMN_NAME	VARCHAR(128)	Name of a procedure parameter.
	Nullable	Contains the null value if the parameter does not have a name.
COLUMN_TYPE	SMALLINT	Type of the parameter: **1** IN **2** INOUT **4** OUT
DATA_TYPE	SMALLINT	The data type of the parameter: **–5** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **-99** CLOB **–95** GRAPHIC **–96** VARGRAPHIC **–350** DBCLOB **–8** NCHAR **–9** NVARCHAR **–10** NCLOB **-98** BLOB **91** DATE **92** TIME **93** TIMESTAMP **70** DATALINK **–100** ROWID **-370** XML **17** DISTINCT **50** ARRAY

Table 215. SQLPROCEDURECOLS view (continued)

Column Name	Data Type	Description
TYPE_NAME	VARCHAR(261)	The name of the data type of the parameter: **BIGINT** BIGINT **INTeger** INTEGER **SMALLINT** SMALLINT **DECIMAL** DECIMAL **NUMERIC** NUMERIC **FLOAT** DOUBLE PRECISION **REAL** REAL **DECFLOAT** DECFLOAT **CHARacter** CHARACTER **CHARacter FOR BIT DATA** CHARACTER FOR BIT DATA **VARCHAR** VARCHAR **VARCHAR FOR BIT DATA** VARCHAR FOR BIT DATA **CLOB** CLOB **GRAPHIC** GRAPHIC **VARGRAPHIC** VARGRAPHIC **DBCLOB** DBCLOB **NCHAR** NCHAR **NVARCHAR** NVARCHAR **NCLOB** NCLOB **BINARY** BINARY **VARBINARY** VARBINARY **BLOB** BLOB **DATE** DATE **TIME** TIME **TIMESTAMP** TIMESTAMP **DATALINK** DATALINK **ROWID** ROWID **XML** XML **Qualified Type Name** DISTINCT **Array Type Name** ARRAY
COLUMN_SIZE	INTEGER	Length of the parameter.
BUFFER_LENGTH	INTEGER	Indicates the length of the parameter in a buffer.

Table 215. SQLPROCEDURECOLS view (continued)

Column Name	Data Type	Description
DECIMAL_DIGITS	SMALLINT	Scale of numeric or datetime data.
	Nullable	Contains the null value if the parameter is not decimal, numeric, binary, time or timestamp.
NUM_PREC_RADIX	SMALLINT	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits:
	Nullable	**2** Binary; floating-point precision is specified in binary digits.
		10 Decimal; all other numeric types are specified in decimal digits.
		Contains the null value if the parameter is not numeric.
NULLABLE	SMALLINT	Indicates whether the parameter is nullable.
		0 The parameter does not allow nulls.
		1 The parameter does allow nulls.
REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement.
		Contains the null value if there is no long comment.
	Nullable	
COLUMN_DEF	DBCLOB(64K)	The expression string used to calculate the default value of a parameter, if one exists. If the default value is the null value, the expression string is the keyword NULL. Contains the null value if the parameter has no default.
	CCSID(1200) Nullable	

Table 215. SQLPROCEDURECOLS view (continued)

Column Name	Data Type	Description
SQL_DATA_TYPE	SMALLINT	The SQL data type of the parameter: **–5** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **–99** CLOB **–95** GRAPHIC **–96** VARGRAPHIC **–350** DBCLOB **–8** NCHAR **–9** NVARCHAR **–10** NCLOB **–98** BLOB **9** DATE **9** TIME **9** TIMESTAMP **70** DATALINK **–100** ROWID **-370** XML **17** DISTINCT **50** ARRAY
SQL_DATETIME_SUB	SMALLINT Nullable	The datetime subtype of the parameter: **1** DATE **2** TIME **3** TIMESTAMP Contains the null value if the data type is not a datetime data type.
CHAR_OCTET_LENGTH	INTEGER Nullable	Indicates the length in characters of the parameter. Contains the null value if the column is not a string.
ORDINAL_POSITION	INTEGER	Numeric place of the parameter in the parameter list, ordered from left to right.
IS_NULLABLE	VARCHAR(3)	Indicates whether the parameter is nullable. **NO** The parameter does not allow nulls. **YES** The parameter does allow nulls.
SPECIFIC_NAME	VARCHAR(128)	Specific name of the procedure instance.

Table 215. SQLPROCEDURECOLS view (continued)

Column Name	Data Type	Description	
JDBC_DATA_TYPE	INTEGER	The JDBC data type of the parameter:	
		–5	BIGINT
		4	INTEGER
		5	SMALLINT
		3	DECIMAL
		2	NUMERIC
		8	DOUBLE PRECISION
		7	REAL
		1111	DECFLOAT
		1	CHARACTER or GRAPHIC
		-15	NCHAR
		–2	CHARACTER FOR BIT DATA or BINARY
		12	VARCHAR or VARGRAPHIC
		-9	NVARCHAR
		–3	VARCHAR FOR BIT DATA or VARBINARY
		2005	CLOB or DBCLOB
		2011	NCLOB
		2004	BLOB
		91	DATE
		92	TIME
		93	TIMESTAMP
		70	DATALINK
		-8	ROWID
		2009	XML
		2001	DISTINCT
		2003	ARRAY

Table 215. SQLPROCEDURECOLS view (continued)

Column Name	Data Type	Description
I_DATA_TYPE	INTEGER	Indicates the IBM i CLI data type of the parameter. **19** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **14** CLOB **95** GRAPHIC or NCHAR **96** VARGRAPHIC or NVARCHAR **15** DBCLOB or NCLOB **13** BLOB **91** DATE **92** TIME **93** TIMESTAMP **16** DATALINK **1111** ROWID **-370** XML **2001** DISTINCT **50** ARRAY
SOURCE_TYPE_NAME	VARCHAR(128) Nullable	If the parameter data type is a user-defined type, the built-in data type name of its source type. Contains the null value if the parameter data type is not a user-defined type.
SOURCE_SQL_DATA_TYPE	SMALLINT Nullable	If the parameter data type is a user-defined type, the built-in SQL_DATA_TYPE of its source type. For values see SQL_DATA_TYPE. Contains the null value if the parameter data type is not a user-defined type.
SOURCE_JDBC_DATA_TYPE	SMALLINT Nullable	If the parameter data type is a user-defined type, the built-in JDBC_DATA_TYPE of its source type. For values see JDBC_DATA_TYPE. Contains the null value if the parameter data type is not a user-defined type.
MAXIMUM_CARDINALITY	BIGINT Nullable	The maximum cardinality of the array data type. Contains the null value if the type is not an array type.

SQLPROCEDURES

The SQLPROCEDURES view contains one row for every procedure.

The following table describes the columns in the view:

Table 216. SQLPROCEDURES view

Column Name	Data Type	Description
PROCEDURE_CAT	VARCHAR(128)	Relational database name
PROCEDURE_SCHEM	VARCHAR(128)	Name of the schema of the procedure instance.
PROCEDURE_NAME	VARCHAR(128)	Name of the procedure.
NUM_INPUT_PARAMS	INTEGER	Identifies the number of input parameters. 0 indicates that there are no input parameters.
NUM_OUTPUT_PARAMS	INTEGER	Identifies the number of output parameters. 0 indicates that there are no output parameters.
NUM_RESULT_SETS	SMALLINT	Identifies the maximum number of result sets returned. 0 indicates that there are no result sets.
REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
PROCEDURE_TYPE	SMALLINT	Reserved. Contains 0.
NUM_INOUT_PARAMS	INTEGER	Identifies the number of input/output parameters. 0 indicates that there are no input/output parameters.
SPECIFIC_NAME	VARCHAR(128)	Specific name of the procedure instance.

SQLSCHEMAS

The SQLSCHEMAS view contains one row for every schema.

The following table describes the columns in the view:

Table 217. SQLSCHEMAS view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the schema.
TABLE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
TABLE_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	Reserved. Contains the null value.
TYPE_CAT	VARCHAR(128) Nullable	Reserved. Contains the null value.
TYPE_SCHEM	VARCHAR(128) Nullable	Reserved. Contains the null value.
TYPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
SELF_REFERENCING_COL_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
REF_GENERATION	VARCHAR(128) Nullable	Reserved. Contains the null value.
DBNAME	VARCHAR(8) Nullable	Reserved. Contains the null value.
SCHEMA_TEXT	VARGRAPHIC(50) CCSID 1200 Nullable	A character string that describes the schema. Contains the empty string if there is no text.
SYSTEM_TABLE_SCHEMA	CHAR(10)	System schema name.

SQLSPECIALCOLUMNS

The SQLSPECIALCOLUMNS view contains one row for every column of a primary key, unique constraint, or unique index that can identify a row of the table.

The following table describes the columns in the view:

Table 218. SQLSPECIALCOLUMNS view

Column Name	Data Type	Description
SCOPE	SMALLINT	Reserved. Contains 0.
COLUMN_NAME	VARCHAR(128)	Column name
DATA_TYPE	SMALLINT	The data type of the column: **–5** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA **12** VARCHAR **–3** VARCHAR FOR BIT DATA **40** CLOB **–95** GRAPHIC **–96** VARGRAPHIC **–350** DBCLOB **–8** NCHAR **–9** NVARCHAR **–10** NCLOB **–2** BINARY **–3** VARBINARY **30** BLOB **91** DATE **92** TIME **93** TIMESTAMP **70** DATALINK **–100** ROWID **17** DISTINCT

Table 218. SQLSPECIALCOLUMNS view (continued)

Column Name	Data Type	Description
TYPE_NAME	VARCHAR(260)	The name of the data type of the parameter: **BIGINT** BIGINT **INTeger** INTEGER **SMALLINT** SMALLINT **DECIMAL** DECIMAL **NUMERIC** NUMERIC **FLOAT** DOUBLE PRECISION **REAL** REAL **DECFLOAT** DECFLOAT **CHARacter** CHARACTER **CHARacter FOR BIT DATA** CHARACTER FOR BIT DATA **VARCHAR** VARCHAR **VARCHAR FOR BIT DATA** VARCHAR FOR BIT DATA **CLOB** CLOB **GRAPHIC** GRAPHIC **VARGRAPHIC** VARGRAPHIC **DBCLOB** DBCLOB **NCHAR** NCHAR **NVARCHAR** NVARCHAR **NCLOB** NCLOB **BINARY** BINARY **VARBINARY** VARBINARY **BLOB** BLOB **DATE** DATE **TIME** TIME **TIMESTAMP** TIMESTAMP **DATALINK** DATALINK **ROWID** ROWID **XML** XML **Qualified Type Name** DISTINCT
COLUMN_SIZE	INTEGER	The length of the column.
BUFFER_LENGTH	INTEGER	Indicates the length of the column in a buffer.
DECIMAL_DIGITS	SMALLINT	Indicates the number of digits for a numeric column.
	Nullable	Contains the null value if the column is not numeric.

Table 218. SQLSPECIALCOLUMNS view (continued)

Column Name	Data Type	Description	
PSEUDO_COLUMN	SMALLINT	Indicates whether this is a ROWID, identity, or row change timestamp column.	
		1	The column is not a ROWID, identity, or row change timestamp column.
		2	The column is a ROWID, identity, or row change timestamp column.
TABLE_CAT	VARCHAR(128)	Relational database name	
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema that contains the table.	
TABLE_NAME	VARCHAR(128)	Name of the table.	
NULLABLE	SMALLINT	Indicates whether the column can contain the null value.	
		0	The column is not nullable.
		1	The column is nullable.
JDBC_DATA_TYPE	SMALLINT	Indicates the JDBC data type of the column.	
		–5	BIGINT
		4	INTEGER
		5	SMALLINT
		3	DECIMAL
		2	NUMERIC
		8	DOUBLE PRECISION
		7	REAL
		1111	DECFLOAT
		1	CHARACTER or GRAPHIC
		-15	NCHAR
		–2	CHARACTER FOR BIT DATA or BINARY
		12	VARCHAR or VARGRAPHIC
		-9	NVARCHAR
		–3	VARCHAR FOR BIT DATA or VARBINARY
		2005	CLOB or DBCLOB
		2011	NCLOB
		2004	BLOB
		91	DATE
		92	TIME
		93	TIMESTAMP
		70	DATALINK
		1111	ROWID
		2009	XML
		2001	DISTINCT

Table 218. SQLSPECIALCOLUMNS view (continued)

Column Name	Data Type	Description
I_DATA_TYPE	SMALLINT	Indicates the IBM i CLI data type of the column. **19** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **14** CLOB **95** GRAPHIC or NCHAR **96** VARGRAPHIC or NVARCHAR **15** DBCLOB or NCLOB **13** BLOB **91** DATE **92** TIME **93** TIMESTAMP **16** DATALINK **1111** ROWID **-370** XML **2001** DISTINCT
SOURCE_TYPE_NAME	VARCHAR(128) Nullable	If the column data type is a user-defined type, the built-in data type name of its source type. Contains the null value if the column data type is not a user-defined type.
SOURCE_SQL_DATA_TYPE	SMALLINT Nullable	If the column data type is a user-defined type, the built-in SQL_DATA_TYPE of its source type. For values see SQL_DATA_TYPE. Contains the null value if the column data type is not a user-defined type.
SOURCE_JDBC_DATA_TYPE	SMALLINT Nullable	If the column data type is a user-defined type, the built-in JDBC_DATA_TYPE of its source type. For values see JDBC_DATA_TYPE. Contains the null value if the column data type is not a user-defined type.

SQLSTATISTICS

The SQLSTATISTICS view contains statistic information about a table.

The following table describes the columns in the view:

Table 219. SQLSTATISTICS view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema of the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
NON_UNIQUE	SMALLINT Nullable	Indicates whether an index prohibits duplicate keys on the table. Contains the null value if the TYPE is 0.
INDEX_QUALIFIER	VARCHAR(128) Nullable	Name of the schema of the index. Contains the null value if the TYPE is 0.
INDEX_NAME	VARCHAR(128) Nullable	Name of the index. Contains the null value if the TYPE is 0.
TYPE	SMALLINT	Indicates the type of information returned: **0** The number of rows in the table. **3** An index on the table.
ORDINAL_POSITION	SMALLINT Nullable	Indicates the ordinal position of the key in the index. Contains the null value if the TYPE is 0.
COLUMN_NAME	DBCLOB(2097152) CCSID 1200 Nullable	If the key column is an expression, contains the expression. Contains the column name if the key column is not an expression. Contains the null value if the TYPE is 0.
ASC_OR_DESC	CHAR(1) Nullable	Order of the column in the key: **A** Ascending **D** Descending Contains the null value if the TYPE is 0.
CARDINALITY	BIGINT Nullable	Reserved. Contains the null value.
PAGES	BIGINT Nullable	Reserved. Contains the null value.
FILTER_CONDITION	DBCLOB(2097152) CCSID 1200 Nullable	Indicates whether the index is a sparse index. **search-condition** This is an SQL index with a WHERE clause. **empty-string** This is a DDS-created select/omit index. Contains the null value if the TYPE is 0 or this is not a sparse index.
I_INDEXTYPE	INTEGER	Contains the type of the index. **0** The TYPE is 0. **1** The index is a PRIMARY KEY or keyed physical file. **2** The index is an SQL index. **3** The index is a keyed logical file. **4** The index is a UNIQUE or REFERENTIAL constraint.

SQLTABLEPRIVILEGES

The SQLTABLEPRIVILEGES view contains one row for every privilege granted on a table. Note that this catalog view cannot be used to determine whether a user is authorized to a table or view because the privilege to use a table or view could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the view:

Table 220. SQLTABLEPRIVILEGES view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema of the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
PRIVILEGE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the table. **DELETE** The privilege to delete rows from the table. **INDEX** The privilege to create an index on the table. **INSERT** The privilege to insert rows into the table. **REFERENCES** The privilege to reference the table in a referential constraint. **SELECT** The privilege to select rows from the table. **UPDATE** The privilege to update the table.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.
DBNAME	VARCHAR(8) Nullable	Reserved. Contains the null value.

SQLTABLES

The SQLTABLES view contains one row for every table, view, and alias.

The following table describes the columns in the view:

Table 221. SQLTABLES view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the schema containing the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
TABLE_TYPE	VARCHAR(24)	Indicates the type of the table: **ALIAS** The table is an alias. **MATERIALIZED QUERY TABLE** The object is a materialized query table. **SYSTEM TABLE** The table is a system table. **TABLE** The table is an SQL table or physical file. **VIEW** The table is an SQL view or logical file.
REMARKS	VARGRAPHIC(2000) CCSID 1200 Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
TYPE_CAT	VARCHAR(128) Nullable	Reserved. Contains the null value.
TYPE_SCHEM	VARCHAR(128) Nullable	Reserved. Contains the null value.
TYPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
SELF_REF_COL_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
REF_GENERATION	VARCHAR(128) Nullable	Reserved. Contains the null value.
DBNAME	VARCHAR(8) Nullable	Reserved. Contains the null value.
TABLE_TEXT	VARGRAPHIC(50) CCSID 1200	A character string provided with the LABEL statement.

SQLTYPEINFO

The SQLTYPEINFO table contains one row for every built-in data type.

The following table describes the columns in the table:

Table 222. SQLTYPEINFO table

Column Name	Data Type	Description
TYPE_NAME	VARCHAR(128)	Name of the built-in data type: **BIGINT** BIGINT **INTeger** INTEGER **SMALLINT** SMALLINT **DECIMAL** DECIMAL **NUMERIC** NUMERIC **FLOAT** DOUBLE PRECISION **REAL** REAL **DECFLOAT** DECFLOAT **CHARacter** CHARACTER **CHARacter FOR BIT DATA** CHARACTER FOR BIT DATA **VARCHAR** VARCHAR **VARCHAR FOR BIT DATA** VARCHAR FOR BIT DATA **CLOB** CLOB **GRAPHIC** GRAPHIC **VARGRAPHIC** VARGRAPHIC **DBCLOB** DBCLOB **NCHAR** NCHAR **NVARCHAR** NVARCHAR **NCLOB** NCLOB **BINARY** BINARY **VARBINARY** VARBINARY **BLOB** BLOB **DATE** DATE **TIME** TIME **TIMESTAMP** TIMESTAMP **DATALINK** DATALINK **ROWID** ROWID **XML** XML

Table 222. SQLTYPEINFO table (continued)

Column Name	Data Type	Description
DATA_TYPE	SMALLINT	The data type of the built-in data type: **–5** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **-99** CLOB **–95** GRAPHIC **–96** VARGRAPHIC **–350** DBCLOB **–8** NCHAR **–9** NVARCHAR **–10** NCLOB **-99** BLOB **91** DATE **92** TIME **93** TIMESTAMP **70** DATALINK **–100** ROWID **-370** XML **2001** DISTINCT
COLUMN_SIZE	INTEGER Nullable	The maximum length of the data type.
LITERAL_PREFIX	VARCHAR(128) Nullable	Indicates the prefix for a string literal. Contains the null value if the data type is not a string.
LITERAL_SUFFIX	VARCHAR(128) Nullable	Indicates the suffix for a string literal. Contains the null value if the data type is not a string.
CREATE_PARAMS	VARCHAR(128) Nullable	Indicates the parameters supported with the data type. **length** The parameter is a length. Returned for all string data types and DATALINK. **precision,scale** The parameters include precision and scale. Returned for the DECIMAL and NUMERIC data types. Contains the null value for all other data types.
NULLABLE	SMALLINT Nullable	Indicates whether the data type is nullable. **0** The data type does not allow nulls. **1** The data type does allow nulls.

Table 222. SQLTYPEINFO table (continued)

Column Name	Data Type	Description
CASE_SENSITIVE	SMALLINT	Indicates whether the data type is case sensitive.
	Nullable	**0** The data type is not case sensitive. **1** The data type is case sensitive.
SEARCHABLE	SMALLINT	Indicates whether the data type can be used in a predicate.
	Nullable	**0** The data type cannot be used in predicates. **2** The data type can be used in all predicates except the LIKE predicate. **3** The data type can be used in all predicates including the LIKE predicate.
UNSIGNED_ATTRIBUTE	SMALLINT	Indicates whether the numeric data type is signed or unsigned.
	Nullable	**0** The data type is signed. **1** The data type is unsigned. Contains the null value if the data type is not numeric.
FIXED_PREC_SCALE	SMALLINT	Indicates whether the data type has a fixed precision and scale.
	Nullable	**0** The data type does not have a fixed precision and scale. **1** The data type does have a fixed precision and scale. Contains the null value if the data type is not numeric.
AUTO_UNIQUE_VALUE	SMALLINT	Indicates whether the numeric data type is auto-incrementing:
	Nullable	**0** The data type is not auto-incrementing. **1** The data type is auto-incrementing. Contains the null value if the data type is not numeric.
LOCAL_TYPE_NAME	VARCHAR(128)	Reserved. Contains the null value.
	Nullable	
MINIMUM_SCALE	SMALLINT	Indicates the minimum scale of numeric data types.
	Nullable	Contains the null value if the data type is not numeric.
MAXIMUM_SCALE	SMALLINT	Indicates the maximum scale of numeric data types.
	Nullable	Contains the null value if the data type is not numeric.

Table 222. SQLTYPEINFO table (continued)

Column Name	Data Type	Description
SQL_DATA_TYPE	SMALLINT Nullable	Indicates the SQL data type value of the data type: **–5** BIGINT **4** INTEGER **5** SMALLINT **3** DECIMAL **2** NUMERIC **8** DOUBLE PRECISION **7** REAL **-360** DECFLOAT **1** CHARACTER **–2** CHARACTER FOR BIT DATA or BINARY **12** VARCHAR **–3** VARCHAR FOR BIT DATA or VARBINARY **-99** CLOB **–95** GRAPHIC **–96** VARGRAPHIC **–350** DBCLOB **–8** NCHAR **–9** NVARCHAR **–10** NCLOB **-98** BLOB **9** DATE **9** TIME **9** TIMESTAMP **70** DATALINK **–100** ROWID **-370** XML **17** DISTINCT
SQL_DATETIME_SUB	SMALLINT Nullable	The datetime subtype of the data type: **1** DATE **2** TIME **3** TIMESTAMP Contains the null value if the data type is not a datetime data type.
NUM_PREC_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the parameter is not numeric.
INTERVAL_PRECISION	SMALLINT Nullable	Reserved. Contains the null value.

Table 222. SQLTYPEINFO table (continued)

Column Name	Data Type	Description	
JDBC_DATA_TYPE	SMALLINT	The JDBC data type value of the data type:	
		–5	BIGINT
		4	INTEGER
		5	SMALLINT
		3	DECIMAL
		2	NUMERIC
		8	DOUBLE PRECISION
		7	REAL
		1111	DECFLOAT
		1	CHARACTER or GRAPHIC
		-15	NCHAR
		–2	CHARACTER FOR BIT DATA or BINARY
		12	VARCHAR or VARGRAPHIC
		-9	NVARCHAR
		–3	VARCHAR FOR BIT DATA or VARBINARY
		2005	CLOB or DBCLOB
		2011	NCLOB
		2004	BLOB
		91	DATE
		92	TIME
		93	TIMESTAMP
		70	DATALINK
		-8	ROWID
		2009	XML
		2001	DISTINCT

Table 222. SQLTYPEINFO table (continued)

Column Name	Data Type	Description	
I_DATA_TYPE	SMALLINT	Indicates the IBM i CLI data type of the data type.	
		19	BIGINT
		4	INTEGER
		5	SMALLINT
		3	DECIMAL
		2	NUMERIC
		8	DOUBLE PRECISION
		7	REAL
		-360	DECFLOAT
		1	CHARACTER
		–2	CHARACTER FOR BIT DATA or BINARY
		12	VARCHAR
		–3	VARCHAR FOR BIT DATA or VARBINARY
		14	CLOB
		95	GRAPHIC or NCHAR
		96	VARGRAPHIC or NVARCHAR
		15	DBCLOB or NCLOB
		13	BLOB
		91	DATE
		92	TIME
		93	TIMESTAMP
		16	DATALINK
		1111	ROWID
		-370	XML
		2001	DISTINCT

SQLUDTS

The SQLUDTS view contains one row for every distinct type.

The following table describes the columns in the view:

Table 223. SQLUDTS view

Column Name	Data Type	Description
TYPE_CAT	VARCHAR(128)	Relational database name
TYPE_SCHEM	VARCHAR(128)	Name of the schema containing the user-defined type.
TYPE_NAME	VARCHAR(128)	Name of the user-defined type.

Table 223. SQLUDTS view (continued)

Column Name	Data Type	Description
CLASS_NAME	VARCHAR(20)	Java class name of the user-defined type. **java.math.BigInteger** BIGINT **java.lang.Integer** INTEGER **java.lang.Short** SMALLINT **java.math.BigDecimal** DECIMAL **java.sql.BigDecimal** NUMERIC **java.lang.Double** DOUBLE PRECISION **java.lang.Float** REAL **java.math.BigDecimal** DECFLOAT **java.lang.String** CHARACTER **byte[]** CHARACTER FOR BIT DATA **java.lang.String** VARCHAR **byte[]** VARCHAR FOR BIT DATA **java.sql.Clob** CLOB **java.lang.String** GRAPHIC **java.lang.String** VARGRAPHIC **java.sql.Clob** DBCLOB **byte[]** BINARY **byte[]** VARBINARY **java.sql.Blob** BLOB **java.sql.Date** DATE **java.sql.Time** TIME **java.sql.Timestamp** TIMESTAMP **java.net.URL** DATALINK **byte[]** ROWID **java.sql.SQLXML** XML
DATA_TYPE	SMALLINT	Reserved. Contains 2001.

Table 223. SQLUDTS view (continued)

Column Name	Data Type	Description	
BASE_TYPE	SMALLINT	The source data type of the user-defined data type:	
		–5	BIGINT
		4	INTEGER
		5	SMALLINT
		3	DECIMAL
		2	NUMERIC
		8	DOUBLE PRECISION
		7	REAL
		1111	DECFLOAT
		1	CHARACTER or GRAPHIC
		-15	NCHAR
		–2	CHARACTER FOR BIT DATA or BINARY
		12	VARCHAR or VARGRAPHIC
		-9	NVARCHAR
		–3	VARCHAR FOR BIT DATA or VARBINARY
		2005	CLOB or DBCLOB
		2011	NCLOB
		2004	BLOB
		91	DATE
		92	TIME
		93	TIMESTAMP
		70	DATALINK
		1111	ROWID
		2009	XML
REMARKS	VARGRAPHIC(2000) CCSID 1200	A character string supplied with the COMMENT statement.	
		Contains the null value if there is no comment.	
	Nullable		

ANS and ISO catalog views

There are two versions of some of the ANS and ISO catalog views. The version documented is the normal set of ANS and ISO views. A second set of views have names that are limited to no more than 18 characters and other than the view names are not documented in this book.

The ANS and ISO catalog includes the following tables in the QSYS2 library:

View Name	Shorter View Name	Description
"SQL_FEATURES" on page 1790		Information about features supported by the database manager
"SQL_LANGUAGES" on page 1791	SQL_LANGUAGES_S	Information about the supported languages
"SQL_SIZING" on page 1792		Information about the limits supported by the database manager

The ANS and ISO catalog includes the following views and tables in the SYSIBM and QSYS2 libraries:

View Name	Shorter View Name	Description
"AUTHORIZATIONS" on page 1764		Information about authorization IDs
"CHARACTER_SETS" on page 1765	CHARACTER_SETS_S	Information about supported CCSIDs
"CHECK_CONSTRAINTS" on page 1766		Information about check constraints
"COLUMN_PRIVILEGES" on page 1767		Information about column privileges
"COLUMNS" on page 1768	COLUMNS_S	Information about columns
"INFORMATION_SCHEMA_CATALOG_NAME" on page 1772	CATALOG_NAME	Information about the relational database
"PARAMETERS" on page 1773	PARAMETERS_S	Information about procedure parameters
"REFERENTIAL_CONSTRAINTS" on page 1777	REF_CONSTRAINTS	Information about referential constraints
"ROUTINE_PRIVILEGES" on page 1788	RTNPRIV	Information about routine privileges
"ROUTINES" on page 1778	ROUTINES_S	Information about routines
"SCHEMATA" on page 1789	SCHEMATA_S	Statistical information about schemas
"TABLE_CONSTRAINTS" on page 1793		Information about constraints
"TABLE_PRIVILEGES" on page 1794		Information about table privileges
"TABLES" on page 1795	TABLES_S	Information about tables
"UDT_PRIVILEGES" on page 1796	UDTPRIV	Information about type privileges
"USAGE_PRIVILEGES" on page 1797	USAGEPRIV	Information about sequence and XML schema privileges
"USER_DEFINED_TYPES" on page 1798	UDT_S	Information about types
"VARIABLE_PRIVILEGES" on page 1802	VARPRIV	Information about global variable privileges
"VIEWS" on page 1803		Information about views

AUTHORIZATIONS

The AUTHORIZATIONS view contains one row for every authorization ID.

The following table describes the columns in the view:

Table 224. AUTHORIZATIONS view

Column Name	Data Type	Description
AUTHORIZATION_NAME	VARCHAR(128)	Authorization ID name
AUTHORIZATION_TYPE	VARCHAR(4)	The type of authorization ID. Contains 'USER'.
AUTHORIZATION_ATTR	VARCHAR(5)	The type of user profile. Contains 'USER' or 'GROUP'.

CHARACTER_SETS

The CHARACTER_SETS view contains one row for every CCSID supported.

The following table describes the columns in the view:

Table 225. CHARACTER_SETS view

Column Name	Data Type	Description
CHARACTER_SET_CATALOG	VARCHAR(128)	Relational database name
CHARACTER_SET_SCHEMA	VARCHAR(128)	The schema name of the character set. Contains 'SYSIBM'.
CHARACTER_SET_NAME	VARCHAR(128)	The character set name.
FORM_OF_USE	VARCHAR(128) Nullable	Reserved. Contains the null value.
NUMBER_OF_CHARACTERS	INTEGER Nullable	Reserved. Contains the null value.
DEFAULT_COLLATE_CATALOG	VARCHAR(128)	Reserved. Contains the relational database name.
DEFAULT_COLLATE_SCHEMA	VARCHAR(128)	Reserved. Contains SYSIBM.
DEFAULT_COLLATE_NAME	VARCHAR(128)	Reserved. Contains IBMDEFAULT.

CHECK_CONSTRAINTS

The CHECK_CONSTRAINTS view contains one row for every check constraint.

The following table describes the columns in the view:

Table 226. CHECK_CONSTRAINTS view

Column Name	Data Type	Description
CONSTRAINT_CATALOG	VARCHAR(128)	Relational database name
CONSTRAINT_SCHEMA	VARCHAR(128)	Name of the schema containing the constraint
CONSTRAINT_NAME	VARCHAR(128)	Name of the constraint
CHECK_CLAUSE	VARGRAPHIC(2000) CCSID 1200 Nullable	Text of the check constraint clause Contains the null value if the check clause cannot be contained in the column without truncation.

COLUMN_PRIVILEGES

The COLUMN_PRIVILEGES view contains one row for every privilege granted on a column. Note that this catalog view cannot be used to determine whether a user is authorized to a column because the privilege to use a column could be acquired through a group user profile or special authority (such as *ALLOBJ). Furthermore, the privilege to use a column is also acquired through privileges granted on the table.

The following table describes the columns in the view:

Table 227. COLUMN_PRIVILEGES view

Column Name	Data Type	Description
GRANTOR	VARCHAR(128)	Reserved. Contains the null value.
	Nullable	
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
COLUMN_NAME	VARCHAR(128)	Name of the column.
PRIVILEGE_TYPE	VARCHAR(10)	The privilege granted: **UPDATE** The privilege to update the column. **REFERENCES** The privilege to reference the column in a referential constraint.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.

COLUMNS

The COLUMNS view contains one row for every column.

The following table describes the columns in the view:

Table 228. COLUMNS view

Column Name	Data Type	Description
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema containing the table or view
TABLE_NAME	VARCHAR(128)	Name of the table or view that contains the column
COLUMN_NAME	VARCHAR(128)	Name of the column
ORDINAL_POSITION	INTEGER	Numeric place of the column in the table or view, ordered from left to right
COLUMN_DEFAULT	VARGRAPHIC(2000) CCSID 1200 Nullable	The default value of a column, if one exists. If the default value of the column cannot be represented without truncation, then the value of the column is the string 'TRUNCATED'. The default value is stored in character form. The following special values also exist: **CURRENT_DATE** The default value is the current date. **CURRENT_TIME** The default value is the current time. **CURRENT_TIMESTAMP** The default value is the current timestamp. **NULL** The default value is the null value and DEFAULT NULL was explicitly specified. **USER** The default value is the current job user. Contains the null value if: • The column has no default value. For example, if the column has an IDENTITY attribute or is a row ID, or • A DEFAULT value was not explicitly specified.
IS_NULLABLE	VARCHAR(3)	Indicates whether the column can contain null values: **NO** The column cannot contain null values. **YES** The column can contain null values.

Table 228. COLUMNS view (continued)

Column Name	Data Type	Description
DATA_TYPE	VARCHAR(128)	Type of column: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **DOUBLE PRECISION** Double-precision floating point **REAL** Single-precision floating point **DECFLOAT** Decimal floating-point **CHARACTER** Fixed-length character string **CHARACTER VARYING** Varying-length character string **CHARACTER LARGE OBJECT** Character large object string **GRAPHIC** Fixed-length graphic string **GRAPHIC VARYING** Varying-length graphic string **DOUBLE-BYTE CHARACTER LARGE OBJECT** Double-byte character large object string **NATIONAL CHARACTER** National character **NATIONAL CHARACTER VARYING** Varying-length national character **NATIONAL CHARACTER LARGE OBJECT** National character large object **BINARY** Fixed-length binary string **BINARY VARYING** Varying-length binary string **BINARY LARGE OBJECT** Binary large object string **DATE** Date **TIME** Time **TIMESTAMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **USER-DEFINED** Distinct type
CHARACTER_MAXIMUM_LENGTH	INTEGER	Maximum length of the string for binary, character, and graphic string and XML data types.
	Nullable	Contains the null value if the column is not a string.

Table 228. COLUMNS view (continued)

Column Name	Data Type	Description
CHARACTER_OCTET_LENGTH	INTEGER Nullable	Number of bytes for binary, character, and graphic string and XML data types. Contains the null value if the column is not a string.
NUMERIC_PRECISION	INTEGER Nullable	The precision of all numeric columns. **Note:** This column supplies the precision of all numeric data types, including single-and double-precision floating point and decimal floating-point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the column is not numeric.
NUMERIC_PRECISION_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the column is not numeric.
NUMERIC_SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if the column is not decimal, numeric, or binary.
DATETIME_PRECISION	INTEGER Nullable	The fractional part of a date, time, or timestamp. **0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if the column is not a date, time, or timestamp.
INTERVAL_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
INTERVAL_PRECISION	INTEGER Nullable	Reserved. Contains the null value.
CHARACTER_SET_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.
CHARACTER_SET_SCHEMA	VARCHAR(128) Nullable	The schema name of the character set. Contains SYSIBM. Contains the null value if the column is not a string.
CHARACTER_SET_NAME	VARCHAR(128) Nullable	The character set name. Contains the null value if the column is not a string.
COLLATION_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.
COLLATION_SCHEMA	VARCHAR(128) Nullable	The schema of the collation. Contains SYSIBM. Contains the null value if the column is not a string.
COLLATION_NAME	VARCHAR(128) Nullable	The collation name. Contains IBMBINARY. Contains the null value if the column is not a string.
DOMAIN_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.

Table 228. COLUMNS view (continued)

Column Name	Data Type	Description
DOMAIN_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
DOMAIN_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
UDT_CATALOG	VARCHAR(128) Nullable	The relational database name if this is a distinct type. Contains the null value if this is not a distinct type.
UDT_SCHEMA	VARCHAR(128) Nullable	The name of the schema if this is a distinct type. Contains the null value if this is not a distinct type.
UDT_NAME	VARCHAR(128) Nullable	The name of the distinct type. Contains the null value if this is not a distinct type.
SCOPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
MAXIMUM_CARDINALITY	INTEGER Nullable	Reserved. Contains the null value.
DTD_IDENTIFIER	VARCHAR(128) Nullable	A unique internal identifier for the column.
IS_SELF_REFERENCING	VARCHAR(3)	Reserved. Contains 'NO'.

INFORMATION_SCHEMA_CATALOG_NAME

The INFORMATION_SCHEMA_CATALOG_NAME view contains one row for the relational database.

The following table describes the columns in the view:

Table 229. INFORMATION_SCHEMA_CATALOG_NAME view

Column Name	Data Type	Description
CATALOG_NAME	VARCHAR(128)	Relational database name

PARAMETERS

The PARAMETERS view contains one row for each parameter of a routine in the relational database.

The following table describes the columns in the view:

Table 230. PARAMETERS view

Column Name	Data Type	Description
SPECIFIC_CATALOG	VARCHAR(128)	Relational database name
SPECIFIC_SCHEMA	VARCHAR(128)	Schema name of the routine instance
SPECIFIC_NAME	VARCHAR(128)	Specific name of the routine instance
ORDINAL_POSITION	INTEGER	Numeric place of the parameter in the parameter list, ordered from left to right.
PARAMETER_MODE	VARCHAR(5)	The type of the parameter: **IN** This is an input parameter. **OUT** This is an output parameter. **INOUT** This is an input/output parameter.
IS_RESULT	VARCHAR(3)	Reserved. Contains 'NO'.
AS_LOCATOR	VARCHAR(3)	Indicates whether the parameter was specified as a locator. **NO** The parameter was not specified as a locator. **YES** The parameter was specified as a locator.
PARAMETER_NAME	VARCHAR(128) Nullable	The name of the parameter Contains the null value if the parameter does not have a name.
FROM_SQL_SPECIFIC_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
FROM_SQL_SPECIFIC_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
FROM_SQL_SPECIFIC_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.

Table 230. PARAMETERS view (continued)

Column Name	Data Type	Description
DATA_TYPE	VARCHAR(128) Nullable	Type of the parameter: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **DOUBLE PRECISION** Floating point; DOUBLE PRECISION **REAL** Floating point; REAL **DECFLOAT** Decimal floating-point **CHARACTER** Fixed-length character string **CHARACTER VARYING** Varying-length character string **CHARACTER LARGE OBJECT** Character large object string **GRAPHIC** Fixed-length graphic string **GRAPHIC VARYING** Varying-length graphic string **DOUBLE-BYTE CHARACTER LARGE OBJECT** Double-byte character large object string **NATIONAL CHARACTER** National character **NATIONAL CHARACTER VARYING** Varying-length national character **NATIONAL CHARACTER LARGE OBJECT** National character large object **BINARY** Fixed-length binary string **BINARY VARYING** Varying-length binary string **BINARY LARGE OBJECT** Binary large object string **DATE** Date **TIME** Time **TIMESTAMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **USER-DEFINED** Distinct type or array type
CHARACTER_MAXIMUM_LENGTH	INTEGER Nullable	Maximum length of the string for binary, character, and graphic string and XML data types. Contains the null value if the parameter is not a string.

Table 230. PARAMETERS view (continued)

Column Name	Data Type	Description
CHARACTER_OCTET_LENGTH	INTEGER Nullable	Number of bytes for binary, character, and graphic string and XML data types. Contains the null value if the parameter is not a string.
CHARACTER_SET_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.
CHARACTER_SET_SCHEMA	VARCHAR(128) Nullable	The schema name of the character set. Contains 'SYSIBM'. Contains the null value if the column is not a string.
CHARACTER_SET_NAME	VARCHAR(128) Nullable	The character set name. Contains the null value if the column is not a string.
COLLATION_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.
COLLATION_SCHEMA	VARCHAR(128) Nullable	The schema of the collation. SYSIBM is returned. Contains the null value if the column is not a string.
COLLATION_NAME	VARCHAR(128) Nullable	The collation name. IBMBINARY is returned. Contains the null value if the column is not a string.
NUMERIC_PRECISION	INTEGER Nullable	The precision of all numeric parameters. **Note:** This column supplies the precision of all numeric data types, including single-and double-precision floating point and decimal floating-point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the parameter is not numeric.
NUMERIC_PRECISION_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the parameter is not numeric.
NUMERIC_SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if not decimal, numeric, or binary parameter.
DATETIME_PRECISION	INTEGER Nullable	The fractional part of a date, time, or timestamp. **0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if the parameter is not a date, time, or timestamp.
INTERVAL_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
INTERVAL_PRECISION	INTEGER Nullable	Reserved. Contains the null value.
UDT_CATALOG	VARCHAR(128) Nullable	The relational database name if this is a distinct type or array type.. Contains the null value if this is not a distinct type or array type.

Table 230. PARAMETERS view (continued)

Column Name	Data Type	Description
UDT_SCHEMA	VARCHAR(128) Nullable	The name of the schema if this is a distinct type or array type. Contains the null value if this is not a distinct type or array type.
UDT_NAME	VARCHAR(128) Nullable	The name of the distinct type or array type. Contains the null value if this is not a distinct type or array type.
SCOPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
MAXIMUM_CARDINALITY	BIGINT Nullable	The maximum cardinality of the array type if this parameter is an array type. Contains the null value if this parameter is not an array type.
DTD_IDENTIFIER	VARCHAR(128) Nullable	A unique internal identifier for the parameter.
PARAMETER_DEFAULT	DBCLOB(64K) CCSID 1200 Nullable	The expression string used to calculate the default value of a parameter, if one exists. If the default value is the null value, the expression string is the keyword NULL. Contains the null value if the parameter has no default.

REFERENTIAL_CONSTRAINTS

The REFERENTIAL_CONSTRAINTS view contains one row for each referential constraint.

The following table describes the columns in the view:

Table 231. REFERENTIAL_CONSTRAINTS view

Column Name	Data Type	Description
CONSTRAINT_CATALOG	VARCHAR(128)	Relational database name
CONSTRAINT_SCHEMA	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	VARCHAR(128)	Name of the constraint.
UNIQUE_CONSTRAINT_CATALOG	VARCHAR(128)	Relational database name containing the unique constraint referenced by the referential constraint.
UNIQUE_CONSTRAINT_SCHEMA	VARCHAR(128)	Name of the SQL schema containing the unique constraint referenced by the referential constraint.
UNIQUE_CONSTRAINT_NAME	VARCHAR(128)	Name of the unique constraint referenced by the referential constraint.
MATCH_OPTION	VARCHAR(7)	Reserved. Contains 'NONE'.
UPDATE_RULE	VARCHAR(11)	Update Rule. • NO ACTION • RESTRICT
DELETE_RULE	VARCHAR(11)	Delete Rule • NO ACTION • CASCADE • SET NULL • SET DEFAULT • RESTRICT
COLUMN_COUNT	INTEGER	Count of columns in the constraint.

ROUTINES

The ROUTINES view contains one row for each routine.

The following table describes the columns in the view:

Table 232. ROUTINES view

Column Name	Data Type	Description
SPECIFIC_CATALOG	VARCHAR(128)	Relational database name
SPECIFIC_SCHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	VARCHAR(128)	Specific name of the routine.
ROUTINE_CATALOG	VARCHAR(128)	Relational database name
ROUTINE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the routine.
ROUTINE_NAME	VARCHAR(128)	Name of the routine.
ROUTINE_TYPE	VARCHAR(15)	Type of the routine. **PROCEDURE** This is a procedure. **FUNCTION** This is a function. **INSTANCE METHOD** This is a built-in data type function created for a distinct type.
MODULE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
MODULE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
MODULE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
UDT_CATALOG	VARCHAR(128) Nullable	Relational database name. Contains the null value if this is not an INSTANCE METHOD.
UDT_SCHEMA	VARCHAR(128) Nullable	Name of the SQL schema that contains the distinct type related to this function. Contains the null value if this is not an INSTANCE METHOD.
UDT_NAME	VARCHAR(128) Nullable	Name of the distinct type name related to this function. Contains the null value if this is not an INSTANCE METHOD.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
DATA_TYPE	VARCHAR(128) Nullable	Type of the result of the function: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **DOUBLE PRECISION** Floating point; DOUBLE PRECISION **REAL** Floating point; REAL **DECFLOAT** Decimal floating-point **CHARACTER** Fixed-length character string **CHARACTER VARYING** Varying-length character string **CHARACTER LARGE OBJECT** Character large object string **GRAPHIC** Fixed-length graphic string **GRAPHIC VARYING** Varying-length graphic string **DOUBLE-BYTE CHARACTER LARGE OBJECT** Double-byte character large object string **NATIONAL CHARACTER** National character **NATIONAL CHARACTER VARYING** Varying-length national character **NATIONAL CHARACTER LARGE OBJECT** National character large object **BINARY** Fixed-length binary string **BINARY VARYING** Varying-length binary string **BINARY LARGE OBJECT** Binary large object string **DATE** Date **TIME** Time **TIMESTAMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **USER-DEFINED** Distinct type or Array type Contains the null value if this is not a scalar function.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
CHARACTER_MAXIMUM_LENGTH	INTEGER Nullable	Maximum length of the result string of the function for binary, character, and graphic string and XML data types. Contains the null value if this is not a scalar function or the parameter is not a string.
CHARACTER_OCTET_LENGTH	INTEGER Nullable	Number of bytes for the result string of the function for binary, character, and graphic string and XML data types. Contains the null value if this is not a scalar function or the parameter is not a string.
CHARACTER_SET_CATALOG	VARCHAR(128) Nullable	Relational database name of the result of the function. Contains the null value if this is not a scalar function or the result is not a string.
CHARACTER_SET_SCHEMA	VARCHAR(128) Nullable	The schema name of the character set of the result of the function. Contains 'SYSIBM'. Contains the null value if this is not a scalar function or the result is not a string.
CHARACTER_SET_NAME	VARCHAR(128) Nullable	The character set name of the result of the function. Contains the null value if this is not a scalar function or the result is not a string.
COLLATION_CATALOG	VARCHAR(128) Nullable	Relational database name of the result of the function. Contains the null value if this is not a scalar function or the result is not a string.
COLLATION_SCHEMA	VARCHAR(128) Nullable	The schema of the collation of the result of the function. SYSIBM is returned. Contains the null value if this is not a scalar function or the result is not a string.
COLLATION_NAME	VARCHAR(128) Nullable	The collation name of the result of the function. IBMBINARY is returned. Contains the null value if this is not a scalar function or the result is not a string.
NUMERIC_PRECISION	INTEGER Nullable	The precision of the result of the function. **Note:** This column supplies the precision of all numeric data types, including single-and double-precision floating point and decimal floating-point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if this is not a scalar function or the result is not numeric.
NUMERIC_PRECISION_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if this is not a scalar function or the result is not numeric.
NUMERIC_SCALE	INTEGER Nullable	Scale of numeric result of the function. Contains the null value if this is not a scalar function or the result is not numeric.
DATETIME_PRECISION	INTEGER Nullable	The fractional part of a date, time, or timestamp result of the function. **0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if this is not a scalar function or the result is not a date, time, or timestamp.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
INTERVAL_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
INTERVAL_PRECISION	INTEGER Nullable	Reserved. Contains the null value.
TYPE_UDT_CATALOG	VARCHAR(128) Nullable	The relational database name if the result of the function is a distinct type or array type. Contains the null value if this is not a scalar function or the result is not a distinct type or array type.
TYPE_UDT_SCHEMA	VARCHAR(128) Nullable	The name of the schema if the result of the function is a distinct type or array type. Contains the null value if this is not a scalar function or the result is not a distinct type or array type.
TYPE_UDT_NAME	VARCHAR(128) Nullable	The name of the distinct type if the result of the function is a distinct type or array type. Contains the null value if this is not a scalar function or the result is not a distinct type or array type.
SCOPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
MAXIMUM_CARDINALITY	BIGINT Nullable	The maximum cardinality of the array type if the result of the function is an array type. Contains the null value is this is not an array type.
DTD_IDENTIFIER	VARCHAR(128) Nullable	A unique internal identifier for the result of the function.
ROUTINE_BODY	VARCHAR(8)	The type of the routine body: **EXTERNAL** This is an external routine. **SQL** This is an SQL routine.
ROUTINE_DEFINITION	DBCLOB(2M) CCSID 13488 Nullable	If this is an SQL routine, this column contains the SQL routine body. If this is an obfuscated routine, the text starts with the WRAPPED keyword and is followed by the encoded form of the statement text. Contains the null value if this is not an SQL routine or if the routine body cannot be contained in this column without truncation.
EXTERNAL_NAME	VARCHAR(279) Nullable	If this is an external routine, this column identifies the external program name. • For REXX, the external program name is *schema-name/source-file-name(member-name)*. • For ILE service programs, the external program name is *schema-name/service-program-name(entry-point-name)*. • For Java programs, the external program name is an optional jar-id followed by a *fully-qualified-class-name!method-name* or *fully-qualified-class-name.method-name*. • For all other languages, the external program name is *schema-name/program-name*. Contains the null value if this is a system-generated function or a function sourced on a built-in function.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
EXTERNAL_LANGUAGE	VARCHAR(8) Nullable	If this is an external routine, this column identifies the external program name. **C** The external program is written in C. **C++** The external program is written in C++. **CL** The external program is written in CL. **COBOL** The external program is written in COBOL. **COBOLLE** The external program is written in ILE COBOL. **FORTRAN** The external program is written in FORTRAN. **JAVA** The external program is written in JAVA. **PLI** The external program is written in PL/I. **REXX** The external program is a REXX procedure. **RPG** The external program is written in RPG. **RPGLE** The external program is written in ILE RPG. Contains the null value if this is not an external routine.
PARAMETER_STYLE	VARCHAR(18) Nullable	If this is an external routine, this column identifies the parameter style (calling convention). **DB2GENERAL** This is the DB2GENERAL calling convention. **DB2SQL** This is the DB2SQL calling convention. **GENERAL** This is the GENERAL calling convention. **JAVA** This is the JAVA calling convention. **GENERAL WITH NULLS** This is the GENERAL WITH NULLS calling convention. **SQL** This is the SQL standard calling convention. Contains the null value if this is not an external routine.
IS_DETERMINISTIC	VARCHAR(3)	This column identifies whether the routine is deterministic. That is, whether a call to the routine with the same arguments will always return the same result. **NO** The routine is not deterministic. **YES** The routine is deterministic.
SQL_DATA_ACCESS	VARCHAR(17)	This column identifies whether a routine contains SQL and whether it reads or modifies data. **NO SQL** The routine does not contain any SQL statements. **CONTAINS SQL** The routine contains SQL statements. **READS SQL DATA** The routine possibly reads data from a table or view. **MODIFIES SQL DATA** The routine possibly modifies data in a table or view or issues SQL DDL statements.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
IS_NULL_CALL	VARCHAR(3) Nullable	Identifies whether the function needs to be called if an input parameter is the null value. **NO** This function need not be called if an input parameter is the null value. If this is a scalar function, the result of the function is implicitly null if any of the operands are null. If this is a table function, the result of the function is an empty table if any of the operands are the null value. **YES** This function must be called even if an input operand is null. Contains the null value if this is not a function.
SQL_PATH	VARCHAR(3483) Nullable	If this is an SQL routine, this column identifies the path. Contains the null value if this is not an SQL routine.
SCHEMA_LEVEL_ROUTINE	VARCHAR(3)	Reserved. Contains 'YES'.
MAX_DYNAMIC_RESULT_SETS	SMALLINT	Identifies the maximum number of result sets returned. 0 indicates that there are no result sets.
IS_USER_DEFINED_CAST	VARCHAR(3) Nullable	Identifies whether the this function is a cast function created when a distinct type was created. **NO** This function is not a cast function. **YES** This function is a cast function. Contains the null value if the routine is not a function.
IS_IMPLICITLY_INVOCABLE	VARCHAR(3) Nullable	Identifies whether the this function is a cast function created when a distinct type was created and can be implicitly invoked. **NO** This function is not a cast function. **YES** This function is a cast function and can be implicitly invoked. Contains the null value if the routine is not a function.
SECURITY_TYPE	VARCHAR(22) Nullable	Reserved. Contains 'IMPLEMENTATION DEFINED' if this is an external routine. Contains the null value if the routine is not an external routine.
TO_SQL_SPECIFIC_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
AS_LOCATOR	VARCHAR(3) Nullable	Indicates whether the result was specified as a locator. **NO** The parameter was not specified as a locator. **YES** The parameter was specified as a locator. Contains the null value if this is not a scalar function.
CREATED	TIMESTAMP	Identifies the timestamp when the routine was created.
LAST_ALTERED	TIMESTAMP Nullable	Timestamp when routine was last altered. Contains null if the routine has never been altered.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
NEW_SAVEPOINT_LEVEL	VARCHAR(3) Nullable	Indicates whether the routine starts a new savepoint level. **NO** A new savepoint level is not started when the procedure is called. **YES** A new savepoint level is started when the procedure is called. Contains the null value if this is not a function.
IS_UDT_DEPENDENT	VARCHAR(3)	Indicates whether the routine is dependent on a UDT. **NO** The routine is not dependent on a UDT. **YES** The routine is dependent on a UDT.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
RESULT_CAST_FROM_DATA_TYPE	VARCHAR(128) Nullable	Type of the parameter: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **DOUBLE PRECISION** Floating point; DOUBLE PRECISION **REAL** Floating point; REAL **DECFLOAT** Decimal floating-point **CHARACTER** Fixed-length character string **CHARACTER VARYING** Varying-length character string **CHARACTER LARGE OBJECT** Character large object string **GRAPHIC** Fixed-length graphic string **GRAPHIC VARYING** Varying-length graphic string **DOUBLE-BYTE CHARACTER LARGE OBJECT** Double-byte character large object string **NATIONAL CHARACTER** National character **NATIONAL CHARACTER VARYING** Varying-length national character **NATIONAL CHARACTER LARGE OBJECT** National character large object **BINARY** Fixed-length binary string **BINARY VARYING** Varying-length binary string **BINARY LARGE OBJECT** Binary large object string **DATE** Date **TIME** Time **TIMESTAMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **USER-DEFINED** Distinct Type
RESULT_CAST_AS_LOCATOR	VARCHAR(3) Nullable	Indicates whether the result is cast from a locator. **NO** The result is not cast from a locator. **YES** The result is cast from a locator.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
RESULT_CAST_CHAR_MAX_LENGTH	INTEGER Nullable	Maximum length of the string for binary, character, and graphic string and XML data types. Contains the null value if the parameter is not a string.
RESULT_CAST_CHAR_OCTET_LENGTH	INTEGER Nullable	Number of bytes for binary, character, and graphic string and XML data types. Contains the null value if the parameter is not a string.
RESULT_CAST_CHAR_SET_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.
RESULT_CAST_CHAR_SET_SCHEMA	VARCHAR(128) Nullable	The schema name of the character set. Contains 'SYSIBM'. Contains the null value if the column is not a string.
RESULT_CAST_CHAR_SET_NAME	VARCHAR(128) Nullable	The character set name. Contains the null value if the column is not a string.
RESULT_CAST_COLLATION_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.
RESULT_CAST_COLLATION_SCHEMA	VARCHAR(128) Nullable	The schema of the collation. SYSIBM is returned. Contains the null value if the column is not a string.
RESULT_CAST_COLLATION_NAME	VARCHAR(128) Nullable	The collation name. IBMBINARY is returned. Contains the null value if the column is not a string.
RESULT_CAST_NUMERIC_PRECISION	INTEGER Nullable	The precision of all numeric parameters. **Note:** This column supplies the precision of all numeric data types, including single-and double-precision floating point and decimal floating-point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the parameter is not numeric.
RESULT_CAST_NUMERIC_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits. Contains the null value if the parameter is not numeric.
RESULT_CAST_NUMERIC_SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if not decimal, numeric, or binary parameter.
RESULT_CAST_DATETIME_PRECISION	INTEGER Nullable	The fractional part of a date, time, or timestamp. **0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds). Contains the null value if the parameter is not a date, time, or timestamp.
RESULT_CAST_INTERVAL_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
RESULT_CAST_INTERVAL_PRECISION	INTEGER Nullable	Reserved. Contains the null value.
RESULT_CAST_TYPE_UDT_CATALOG	VARCHAR(128) Nullable	The relational database name if this is a distinct type. Contains the null value if this is not a distinct type.

Table 232. ROUTINES view (continued)

Column Name	Data Type	Description
RESULT_CAST_TYPE_UDT_SCHEMA	VARCHAR(128) Nullable	The name of the schema if this is a distinct type. Contains the null value if this is not a distinct type.
RESULT_CAST_TYPE_UDT_NAME	VARCHAR(128) Nullable	The name of the distinct type Contains the null value if this is not a distinct type.
RESULT_CAST_SCOPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
RESULT_CAST_SCOPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
RESULT_CAST_SCOPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
RESULT_CAST_MAX_CARDINALITY	INTEGER Nullable	Reserved. Contains the null value.
RESULT_CAST_DTD_IDENTIFIER	VARCHAR(128) Nullable	A unique internal identifier for the parameter.

ROUTINE_PRIVILEGES

The ROUTINE_PRIVILEGES view contains one row for every privilege granted on a routine. Note that this catalog view cannot be used to determine whether a user is authorized to a routine because the privilege to use a routine could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the view:

Table 233. ROUTINE_PRIVILEGES view

Column Name	Data Type	Description
GRANTOR	VARCHAR(128)	Reserved. Contains the null value. Nullable
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
SPECIFIC_CATALOG	VARCHAR(128)	Relational database name
SPECIFIC_SCHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	VARCHAR(128)	Specific name of the routine.
ROUTINE_CATALOG	VARCHAR(128)	Relational database name
ROUTINE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the routine.
ROUTINE_NAME	VARCHAR(128)	Name of the routine.
PRIVILEGE_TYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the routine. **EXECUTE** The privilege to execute the routine.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.

SCHEMATA

The SCHEMATA view contains one row for each schema.

The following table describes the columns in the view:

Table 234. SCHEMATA view

Column Name	Data Type	Description
CATALOG_NAME	VARCHAR(128)	Relational database name
SCHEMA_NAME	VARCHAR(128)	Name of the schema
SCHEMA_OWNER	VARCHAR(128)	Owner of the schema
DEFAULT_CHARACTER_SET_CATALOG	VARCHAR(128)	Relational database name
DEFAULT_CHARACTER_SET_SCHEMA	VARCHAR(128)	The schema name of the default character set. Contains 'SYSIBM'.
DEFAULT_CHARACTER_SET_NAME	VARCHAR(128)	The default character set name.
SQL_PATH	VARCHAR(4096)	Reserved. Contains the null value.

Nullable

SQL_FEATURES

The SQL_FEATURES table contains one row for each feature supported by the database manager.

The following table describes the columns in the table:

Table 235. SQL_FEATURES table

Column Name	Data Type	Description
FEATURE_ID	VARCHAR(7) Nullable	ANS and ISO feature ID
FEATURE_NAME	VARCHAR(128)	The name of the ANS and ISO feature.
SUB_FEATURE_ID	VARCHAR(7) Nullable	ANS and ISO subfeature ID
SUB_FEATURE_NAME	VARCHAR(256)	The name of the ANS and ISO subfeature.
IS_SUPPORTED	VARCHAR(3)	Indicates whether the feature is supported: **YES** This feature is supported. **NO** This feature is not supported.
IS_VERIFIED_BY	VARCHAR(128) Nullable	Reserved. Contains the null value.
COMMENTS	VARCHAR(2000) Nullable	Reserved. Contains the null value.

SQL_LANGUAGES

The SQL_LANGUAGES table contains one row for every SQL language binding and programming language for which conformance is claimed.

The following table describes the columns in the SQL_LANGUAGES table:

Table 236. SQL_LANGUAGES table

Column Name	Data Type	Description
SQL_LANGUAGE_SOURCE	VARCHAR(254)	Name of the standard.
SQL_LANGUAGE_YEAR	VARCHAR(254)	Year in which the standard was approved.
SQL_LANGUAGE_CONFORMANCE	VARCHAR(254) Nullable	Level of conformance. **2** For the 1987 and 1989 standards, indicates that Level 2 conformance is claimed. **ENTRY** For the 1992 standard, indicates that Entry Level conformance is claimed. **CORE** For the 1999 standard, indicates that Core Level is conformance is claimed. Contains the null value if conformance is not yet claimed.
SQL_LANGUAGE_INTEGRITY	VARCHAR(254) Nullable	Support of the integrity feature. **YES** conformance is claimed to the integrity feature **NO** conformance is not claimed to the integrity feature Contains the null value if the standard does not have a separate integrity feature.
SQL_LANGUAGE_IMPLEMENTATION	VARCHAR(254) Nullable	Reserved. Contains the null value.
SQL_LANGUAGE_BINDING_STYLE	VARCHAR(254)	The style of binding of the SQL language **EMBEDDED** support for embedded SQL for the language in SQL_LANGUAGE_PROGRAMMING_LANG **DIRECT** DIRECT SQL is supported (for example Interactive SQL) **CLI** Support for CLI for the language in SQL_LANGUAGE_PROGRAMMING_LANG
SQL_LANGUAGE_PROGRAMMING_LANG	VARCHAR(254) Nullable	The language supported by EMBEDDED or CLI. **C** The C language is supported. **COBOL** The COBOL language is supported. **PLI** The PL/I language is supported. Contains the null value if the SQL_LANGUAGE_BINDING_STYLE is DIRECT.

SQL_SIZING

The SQL_SIZING table contains one row for each limit supported by the database manager.

The following table describes the columns in the table:

Table 237. SQL_SIZING table

Column Name	Data Type	Description
SIZING_ID	INTEGER	ANS and ISO sizing ID
SIZING_NAME	VARCHAR(128)	Name of the ANS and ISO sizing.
SUPPORTED_VALUE	BIGINT	Indicates the sizing limit.
	Nullable	Contains the null value if the sizing limit is not applicable.
COMMENTS	VARCHAR(2000)	Reserved. Contains the null value.
	Nullable	

TABLE_CONSTRAINTS

The TABLE_CONSTRAINTS view contains one row for each constraint.

The following table describes the columns in the view:

Table 238. TABLE_CONSTRAINTS view

Column Name	Data Type	Description
CONSTRAINT_CATALOG	VARCHAR(128)	Relational database name
CONSTRAINT_SCHEMA	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	VARCHAR(128)	Name of the constraint.
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the schema containing the table.
TABLE_NAME	VARCHAR(128)	Name of the table which the constraint is created over.
CONSTRAINT_TYPE	VARCHAR(11)	Constraint Type • CHECK • UNIQUE • PRIMARY KEY • FOREIGN KEY
IS_DEFERRABLE	VARCHAR(3)	Indicates whether the constraint checking can be deferred. Contains 'NO'.
INITIALLY_DEFERRED	VARCHAR(3)	Indicates whether the constraint was defined as initially deferred. Contains 'NO'.

TABLE_PRIVILEGES

The TABLE_PRIVILEGES view contains one row for every privilege granted on a table. Note that this catalog view cannot be used to determine whether a user is authorized to a table because the privilege to use a table could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the view:

Table 239. TABLE_PRIVILEGES view

Column Name	Data Type	Description
GRANTOR	VARCHAR(128)	Reserved. Contains the null value. Nullable
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
PRIVILEGE_TYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the table. **DELETE** The privilege to delete rows from the table. **INDEX** The privilege to create an index on the table. **INSERT** The privilege to insert rows into the table. **REFERENCES** The privilege to reference the table in a referential constraint. **SELECT** The privilege to select rows from the table. **UPDATE** The privilege to update the table.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.

TABLES

The TABLES view contains one row for each table, view, and alias.

The following table describes the columns in the view:

Table 240. TABLES view

Column Name	Data Type	Description
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table, view or alias.
TABLE_NAME	VARCHAR(128)	Name of the table, view or alias.
TABLE_TYPE	VARCHAR(24)	Indicates the type of the table: **ALIAS** The table is an alias. **BASE TABLE** The table is an SQL table or physical file. **MATERIALIZED QUERY TABLE** The object is a materialized query table. **VIEW** The table is an SQL view or logical file.
SELF_REFERENCING_COLUMN_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
REFERENCE_GENERATION	VARCHAR(128) Nullable	Reserved. Contains the null value.
USER_DEFINED_TYPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
USER_DEFINED_TYPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
USER_DEFINED_TYPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
IS_INSERTABLE_INTO	VARCHAR(3)	Identifies whether an INSERT is allowed on the table. **NO** An INSERT is not allowed on this table. **YES** An INSERT is allowed on this table.

UDT_PRIVILEGES

The UDT_PRIVILEGES view contains one row for every privilege granted on a type. Note that this catalog view cannot be used to determine whether a user is authorized to a type because the privilege to use a type could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the view:

Table 241. UDT_PRIVILEGES view

Column Name	Data Type	Description
GRANTOR	VARCHAR(128)	Reserved. Contains the null value.
	Nullable	
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
UDT_CATALOG	VARCHAR(128)	Relational database name
UDT_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the type.
UDT_NAME	VARCHAR(128)	Name of the type.
PRIVILEGE_TYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the type. **USAGE** The privilege to use the type.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.

USAGE_PRIVILEGES

The USAGE_PRIVILEGES view contains one row for every privilege granted on a sequence or XML schema. Note that this catalog view cannot be used to determine whether a user is authorized to a sequence or XML schema because the privilege to use a sequence or XML schema could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the view:

Table 242. USAGE_PRIVILEGES view

Column Name	Data Type	Description
GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
OBJECT_CATALOG	VARCHAR(128)	Relational database name
OBJECT_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the object.
OBJECT_NAME	VARCHAR(128)	Name of the object.
OBJECT_TYPE	VARCHAR(10)	The type of the object: **SEQUENCE** The object is a sequence. **XML SCHEMA** The object is an XML schema.
PRIVILEGE_TYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the sequence or XML schema. **USAGE** The privilege to use the sequence or XML schema.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.

USER_DEFINED_TYPES

The USER_DEFINED_TYPES view contains one row for each distinct type.

The following table describes the columns in the view[153]:

Table 243. USER_DEFINED_TYPES view

Column Name	Data Type	Description
USER_DEFINED_TYPE_CATALOG	VARCHAR(128)	Relational database name
USER_DEFINED_TYPE_SCHEMA	VARCHAR(128)	Schema name of the distinct type
USER_DEFINED_TYPE_NAME	VARCHAR(128)	Name of the user that created the distinct type.
USER_DEFINED_TYPE_CATEGORY	VARCHAR(128)	Indicates the type of user-defined type. Contains 'DISTINCT'.
IS_INSTANTIABLE	VARCHAR(3)	Reserved. Contains 'YES'.
IS_FINAL	VARCHAR(3)	Reserved. Contains 'YES'.
ORDERING_FORM	VARCHAR(4)	Indicates what kind of predicates are allowed when this distinct type is a comparand: **FULL** All predicates are allowed. **NONE** No predicates are allowed
ORDERING_CATEGORY	VARCHAR(8)	Reserved. Contains 'MAP'.
ORDERING_ROUTINE_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the ORDERING_FORM is 'NONE'.
ORDERING_ROUTINE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains 'SYSIBM'. Contains the null value if the ORDERING_FORM is 'NONE'.
ORDERING_ROUTINE_NAME	VARCHAR(128) Nullable	Reserved. Contains a data type name. Contains the null value if the ORDERING_FORM is 'NONE'.
REFERENCE_TYPE	VARCHAR(16) Nullable	Reserved. Contains the null value.

153. This view does not contain information about built-in data types.

Table 243. USER_DEFINED_TYPES view (continued)

Column Name	Data Type	Description
DATA_TYPE	VARCHAR(128) Nullable	Source data type of the distinct type: **BIGINT** Big number **INTEGER** Large number **SMALLINT** Small number **DECIMAL** Packed decimal **NUMERIC** Zoned decimal **DOUBLE PRECISION** Floating point; DOUBLE PRECISION **REAL** Floating point; REAL **DECFLOAT** Decimal floating-point **CHARACTER** Fixed-length character string **CHARACTER VARYING** Varying-length character string **CHARACTER LARGE OBJECT** Character large object string **GRAPHIC** Fixed-length graphic string **GRAPHIC VARYING** Varying-length graphic string **DOUBLE-BYTE CHARACTER LARGE OBJECT** Double-byte character large object string **NATIONAL CHARACTER** National character **NATIONAL CHARACTER VARYING** Varying-length national character **NATIONAL CHARACTER LARGE OBJECT** National character large object **BINARY** Fixed-length binary string **BINARY VARYING** Varying-length binary string **BINARY LARGE OBJECT** Binary large object string **DATE** Date **TIME** Time **TIMESTAMP** Timestamp **DATALINK** Datalink **ROWID** Row ID **XML** XML **USER-DEFINED** Distinct Type
CHARACTER_MAXIMUM_LENGTH	INTEGER Nullable	Maximum length of the distinct type for binary, character, and graphic string and XML data types. Contains the null value if the distinct type is not a string.

Table 243. USER_DEFINED_TYPES view (continued)

Column Name	Data Type	Description
CHARACTER_OCTET_LENGTH	INTEGER	Number of bytes of the distinct type for binary, character, and graphic string and XML data types.
	Nullable	Contains the null value if the distinct type is not a string.
CHARACTER_SET_CATALOG	VARCHAR(128)	Relational database name of the distinct type.
	Nullable	Contains the null value if the distinct type is not a string.
CHARACTER_SET_SCHEMA	VARCHAR(128)	The schema name of the character set of the distinct type. Contains 'SYSIBM'.
	Nullable	Contains the null value if the distinct type is not a string.
CHARACTER_SET_NAME	VARCHAR(128)	The character set name of the distinct type.
	Nullable	Contains the null value if the distinct type is not a string.
COLLATION_CATALOG	VARCHAR(128)	Relational database name of the distinct type.
	Nullable	Contains the null value if the distinct type is not a string.
COLLATION_SCHEMA	VARCHAR(128)	The schema of the collation of the distinct type. SYSIBM is returned.
	Nullable	Contains the null value if the distinct type is not a string.
COLLATION_NAME	VARCHAR(128)	The collation name of the distinct type. IBMBINARY is returned.
	Nullable	Contains the null value if the distinct type is not a string.
NUMERIC_PRECISION	INTEGER	The precision of the distinct type. **Note:** This column supplies the precision of all numeric data types, including single-and double-precision floating point and decimal floating-point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits.
	Nullable	Contains the null value if the distinct type is not numeric.
NUMERIC_PRECISION_RADIX	INTEGER	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: **2** Binary; floating-point precision is specified in binary digits. **10** Decimal; all other numeric types are specified in decimal digits.
	Nullable	Contains the null value if the distinct type is not numeric.
NUMERIC_SCALE	SMALLINT	Scale of numeric distinct type.
	Nullable	Contains the null value if the distinct type is not decimal, numeric, or binary.
DATETIME_PRECISION	INTEGER	The fractional part of a date, time, or timestamp distinct type. **0** For DATE and TIME data types **6** For TIMESTAMP data types (number of microseconds).
	Nullable	Contains the null value if the distinct type is not date, time, or timestamp.
INTERVAL_TYPE	VARCHAR(128)	Reserved. Contains the null value.
	Nullable	
INTERVAL_PRECISION	INTEGER	Reserved. Contains the null value.
	Nullable	
SOURCE_DTD_IDENTIFIER	VARCHAR(128)	A unique internal identifier for the source data type.
	Nullable	Contains the null value if the distinct type is not sourced on another distinct type.

Table 243. USER_DEFINED_TYPES view (continued)

Column Name	Data Type	Description
REF_DTD_IDENTIFIER	VARCHAR(256) Nullable	Reserved. Contains the null value.

VARIABLE_PRIVILEGES

The VARIABLE_PRIVILEGES view contains one row for every privilege granted on a global variable. Note that this catalog view cannot be used to determine whether a user is authorized to a global variable because the privilege to use a global variable could be acquired through a group user profile or special authority (such as *ALLOBJ).

The following table describes the columns in the view:

Table 244. VARIABLE_PRIVILEGESS view

Column Name	Data Type	Description
GRANTOR	VARCHAR(128)	Reserved. Contains the null value.
	Nullable	
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
VARIABLE_CATALOG	VARCHAR(128)	Relational database name
VARIABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the global variable.
VARIABLE_NAME	VARCHAR(128)	Name of the global variable.
PRIVILEGE_TYPE	VARCHAR(10)	The privilege granted: **ALTER** The privilege to alter the global variable. **READ** The privilege to read the value of the global variable. **WRITE** The privilege to assign a value to the global variable.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. **NO** The privilege is not grantable. **YES** The privilege is grantable.

VIEWS

The VIEWS view contains one row for each view.

The following table describes the columns in the view:

Table 245. VIEWS view

Column Name	Data Type	Description
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the view.
TABLE_NAME	VARCHAR(128)	Name of the view.
VIEW_DEFINITION	DBCLOB(2M) CCSID 13488 Nullable	The query expression portion of the CREATE VIEW statement.
CHECK_OPTION	VARCHAR(8)	The check option used on the view **NONE** No check option was specified **LOCAL** The local option was specified **CASCADED** The cascaded option was specified
IS_UPDATABLE	VARCHAR(3)	Specifies if the view is updatable: **YES** The view is updatable **NO** The view is read-only

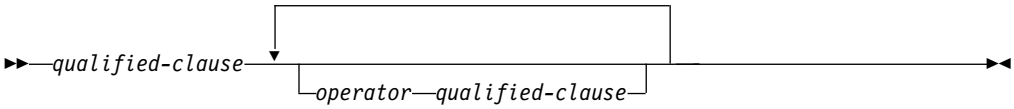
Appendix G. Text search argument syntax

A text search argument is specified when searching for terms in text documents. It consists of search parameters and one or more search terms. The SQL scalar text search functions that use text search arguments are CONTAINS and SCORE.

For information on CONTAINS and SCORE, see "CONTAINS" on page 296 and "SCORE" on page 484.

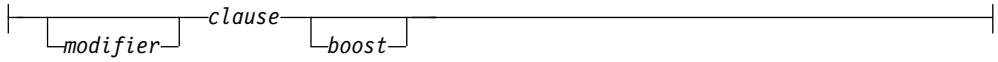
For more information on text search, see OmniFind Text Search Server for DB2 for i

Syntax



operator:

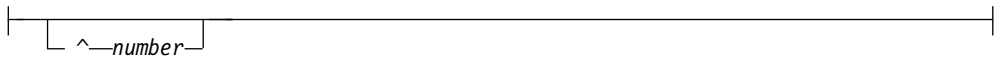
qualified-clause:



modifier:

clause:

boost:



Description

A search argument is a term or sequence of terms, separated by white space, specified when searching in text documents. It consists of one or more search

terms and various optional search parameters.

To perform a simple search, you can enter one or more terms. The search engine returns documents that contain all of those terms or variations of those terms. For example, if you perform a search on the term *king*, documents containing *king* are returned. By default, the search engine also returns variants of the search term. Thus, documents containing *kings* are also returned. Similarly, when you search on two terms, the search engine returns documents containing both terms. If you want the terms to be searched as an exact phrase, simply add quotation marks.

The more specific the search term you use, the more precise the results. However, you may also want to refine your searches by using one or more of the following options:

operator
Specifies whether the search for either or both *qualified clauses* on either side of the *operator* must be satisfied.

AND
The search for both *qualified clauses* on either side of the *operator* must be satisfied.

OR The search for at least one of the *qualified clauses* on either side of the *operator* must be satisfied.

Similar to search conditions in SQL, parentheses can be used to determine which *qualified clauses* and *operators* are evaluated first. If parentheses are not specified, AND is applied before OR.

modifier
Each *clause* can have an occurrence modifier. If a *modifier* is not specified for a *clause*, the default is plus (+).

+ The *clause* is required in the document.

- or NOT
The *clause* must not be specified in the document.

? The *clause* is optional in the document.

clause
Specifies a search string. In the search string, the question mark and asterisk have special meanings. Use a question mark (?) to represent any single character, and an asterisk (*) to represent a string of zero or more characters. If the search string contains multiple words that should be treated as a phrase, include quotation marks (") around the search string. Blanks within a *clause* are ignored.

unqualified-term
An unqualified term is simply a term or a phrase. A term can be a word, such as king; an exact word, such as "king"; or a word that includes a question mark (?) to represent any single character, and an asterisk (*) to represent a string of zero or more characters, such as king* or king?. Similarly, a phrase can be a group of words, such as cabbages and kings; an exact phrase, such as "The King and I"; or a phrase that includes a wildcard, such as "all the king's ho*ses" or "all the king's ?".

If a character in *clause* is one of the characters that has a special meaning in the syntax of the search argument, an escape character (\) can be used to indicate that the subsequent character should be treated as a regular character in the *clause*.

opaque-term
An opaque query term is so called because it is not parsed by the linguistic query parser; opaque terms are identified by their syntax. The opaque term used for text search queries that include XML markup is @xmlxp,, for example: @xmlxp:'/TagA/TagB[. contains("king")]'

boost
boost can be specified for each *clause*. *boost* provides a higher or lower importance to occurrences of the *clause*.

number
Specifies a decimal or integer constant that is greater than 0. If *boost* is not specified for a *clause*, the default boost value is 1.

Notes

Case sensitivity: Searches are not case sensitive, so a search in Spanish for the exact term "DOS" could return documents containing DOS or dos.

Examples: Simple text search

The CONTAINS and SCORE functions can be used to perform a simple text search for a single word or multiple words in a text search index.

The search engine ignores white space between characters. An empty search argument, or one that contains only blanks, does not match anything.

The following table shows some examples of simple text search requests.

Table 246. Simple text search examples

Search word types	Examples	Search results
Single word	`king`	Returns all documents that contain the word `king` or `kings`. The search is not case-sensitive.
Multiple words	`king lear`	Returns all documents that contain `king` and `lear`. The default operator is the logical operator AND.

The operators AND and + (plus sign) are implicit in every text search. For example, the text search for `King Lear` returns the same results as `King AND Lear` or `King + Lear`.

You must enter the logical operators NOT, AND, and OR in all uppercase.

Advanced text search operators

You can use advanced text search operators to refine the search results for the CONTAINS function and the SCORE function.

In the following table, the first column describes the operator that you can use in a text search. (You must enter the logical operators NOT, AND, and OR in all uppercase letters.) The second column shows a sample text search that you might enter. The third column describes the types of results that you might see from the example text search.

Advanced text search operators

Table 247. Advanced search operators and complex text search examples

Operators	Examples	Search results
AND	`"King Lear" AND "Othello"` `"King Lear" "Othello"`	Either text search returns documents that contain both terms `King Lear` and `Othello`. The AND operator is the default conjunction operator. If no logical operator is between the two terms, the AND operator is used. For example, the text search `King Lear` is the same as the text search `King AND Lear`.
OR	`"King Lear" OR Lear`	Returns documents that contain either `King Lear` or just `Lear`. The OR operator links the two terms and finds a matching document if either of the terms exist in a document.
NOT	`"King Lear"` `NOT "Norman Lear"`	Returns documents that contain `King Lear` but not `Norman Lear`. The NOT operator cannot be used with only one term. For example, the following search will return no results: `NOT "King Lear"`.

" "
(Exact match)
	First text search:
	`"King Lear"`
	Second text search:
	`"king"`
		The first text search returns the exact phrase `King Lear`.
		The second text search returns only the word `king` and no other forms, such as `kings` or `kingly`.

* (Wildcard character)	`test*` `te*t`	Returns documents that can match possible combinations, such as `test`, `tests`, and `tester`, or `test` and `text`.
? (Wildcard character)	`test?` `te?t`	Returns documents that can match possible combinations, such as `tests`, or `test` and `text`.

^
(Score boost factor)
some word or phrase^number
	First text search:
	`"King Lear"^4 "Richard III"`
	Second text search:
	`title: (software download)^5`
	`pdf viewer -shipping`
		The first text search forces documents with the phrase `King Lear` to appear higher in the list of search results.
		The second text search forces a document titled `software download` to appear higher in the list of results.
		Although a boost factor must be positive, the boost factor can be less than 1. For example, 0.2. The boost factor number has no limit.

+ (Includes)	`+Lear King`	Returns all documents that contain `Lear` and `King`, which is the same as the text search `Lear AND King`.
- (Excludes)	`"King Lear" -"Lear Jet"`	Returns documents that contain `King Lear` but not `Lear Jet`.
( )	`(King OR Lear) AND plays`	Returns documents that contain either `King` or `Lear` and `plays`. The parentheses ensure that `plays` is found and either term `King` or `Lear` is present.

Table 247. Advanced search operators and complex text search examples (continued)

Operators	Examples	Search results
\ (Escape character)	`\(1\+1\)\:2`	Returns documents that contain `(1+1):2`. Use the \ to clear special characters that are part of the text search syntax. Special characters are + - && || ! ( ) { } [ ] ^ " * ? : \. If a special character is cleared, the special character is analyzed as part of the text search.

Example: Using the CONTAINS function and SCORE function

You can use the CONTAINS function or the SCORE function in the same query to search a text search index and to return if and how frequently the text document matches the search argument criteria.

The example in the following table uses data from the base table BOOKS with the columns ISBN (VARCHAR(20)), ABSTRACT (VARCHAR(10000)), and PRICE (INTEGER).

Table 248. The base table BOOKS

ISBN	ABSTRACT	PRICE
i1	"a b c"	7
i2	"a b d"	10
i3	"a e a"	8

You run the following query:

```
SELECT ISBN, SCORE(ABSTRACT,'"b"')
FROM BOOKS
WHERE CONTAINS (ABSTRACT,'"b"') = 1
```

This query returns the following two rows:

```
i1, 0.3
i3, 0.4
```

The score values might differ depending on the content of the text column.

XML text search

Based on a subset of the XPath language with extensions for text search, XML text search allows you to index and search XML documents so that structural elements can be used separately or can be combined with free text in queries.

Structural elements are tag names, attribute names, and attribute values.

The following list highlights the key features of XML search:

XML structural search
By including special opaque terms in queries, you can search XML documents for structural elements (tag names, attribute names, and attribute values) and text that is scoped by those elements.

XML query tokenization
Free text in XML query terms is tokenized the same way that text in non-XML query terms is tokenized, except that (nested) opaque terms are not supported. Synonyms, wildcard characters, phrases, and lemmatization are supported.

Numeric values
Predicates that compare attribute values to number, date, or dateTime data types are supported.

Complete match
The = (equal sign) operator with a string argument in a predicate calls for a an exact match of all tokens in the string with all tokens in the identified text span. The order is NOT significant.

No UIMA access
Unstructured Information Management Architecture (UIMA) is used for tokenization in XML search, but user-written annotators are not supported.

XML text search grammar

A subset of the XPath language, which is defined by an Extended Backus-Naur Form (EBNF) grammar, is supported by the XML search query parser. Queries that do not conform to the supported grammar are rejected by the query parser, which throws an exception.

The EBNF grammar has been simplified in the following ways by:

- Removing the largest-scale structures for specifying iteration and ranges
- Eliminating filter expressions
- Disallowing absolute pathnames in predicate expressions
- Recognizing only one axis (tag) and only in the forward direction

The following table shows the supported grammar in EBNF notation.

Table 249. Supported query grammar in EBNF notation

Symbol	Production
`XMLQuery ::=`	`QueryPrefix NameSpaceDeclaration QueryString` `  | QueryPrefix QueryString`
`QueryPrefix ::=`	`@xmlxp:`
`QueryString ::=`	`"'" PathExpr "'"`

Table 249. Supported query grammar in EBNF notation (continued)

PathExpr ::=	RelativePathExpr | "/" RelativePathExpr? | "//" RelativePathExpr
RelativePathExpr ::=	StepExpr (("/" | "//") StepExpr)*
StepExpr ::=	("." | AbbrevForwardStep) Predicate?
AbbrevForwardStep ::=	"@"? (QName | "*")
Predicate ::=	"[" PredicateExpr "]"
PredicateExpr ::=	Expr | PredicateExpr ("and" | "or") | "(" PredicateExpr ")"
Expr ::=	ComparisonExpr | ContainmentExpr
ComparisonExpr ::=	PathExpr ComparisonOp Literal
ComparisonOp ::=	"=" | "<" | ">" | "!=" | "<=" | ">="
Literal ::=	StringLiteral | NumericLiteral | DateLiteral
ContainmentExpr ::=	PathExpr "contains" "(" StringLiteral ")" | PathExpr "excludes" "(" StringLiteral ")"
StringLiteral ::=	"\"" [^"]* "\"" | "'" [^']* "'"
DateLiteral ::=	"xs:date(\"" xmlDate "\")" | "xs:dateTime(\"" xmlDateTime "\")"
xmlDate ::=	yyyy"-"mm"-"dd
xmlDateTime ::=	yyyy"-"mm"-"dd [T] hh":"mm":"ss"."uuuuuu
NameSpaceDeclaration ::=	defaultNameSpace (NameSpacePrefixDeclaration)*
defaultNameSpace ::=	"declare default element namespace " StringLiteral ";"
NameSpacePrefixDeclaration ::=	"declare namespace" NameSpacePrefix "=" StringLiteral ";"
NameSpacePrefix ::=	[^":]+

For more information about QName, see http://www.w3.org/TR/REC-xml-names/#NT-QName.

The following information about XML search queries that use XPath notation might not be obvious from the EBNF grammar notations:

- **Names not normalized:** XML tag and attribute names are not normalized when they are indexed. The names are not changed to lowercase or modified in any way. Case is significant in XML tag and attribute names to get a match. Therefore, the strings that are used for XML tag and attribute names in queries must match exactly the names that appear in the source documents.
- **Free text normalization:** Free text in XML documents (text between tags, not inside a tag itself) and attribute values are normalized before indexing. Text in XML search queries (in *contains* or *excludes* operators, or in strings that are surrounded by quotation marks) is normalized, too. Features such as phrases, synonyms, wildcard characters, and lemmas are supported.
- **Operator precedence:** In XML search predicates, containment operators and comparison operators take precedence over logical operators, and all logical operators have the same precedence. Containment operators are *contains* and

excludes. Comparison operators are =, !=, <, >, <= and >=. Logical operators are "and" and "or." You can use parentheses to ensure the desired precedence.

- **Semantics:** In XML search predicates, the comparison operators can be applied only to attribute values and not to tags.

Examples: XPath text search

All valid XPath queries that are sent to the XML parser must be written in a subset of the XPath language using opaque terms. Opaque terms are not parsed by the linguistic text search parser.

The text search parser recognizes an opaque term by the syntax that is used in the text search. For example:

```
@xpath:'query'
```

where *query* is the text shown in the examples in the following table.

Table 250. Examples of valid XPath queries

Query	Description
/sentences	Any document with a top-level tag called *sentences*.
//sentences	Any document with a tag at any level called *sentences*.
/sentence/paragraph	Any document with a top-level tag *sentence* having a direct child tag *paragraph*.
/sentence/paragraph/	Any document with a top-level tag *sentence* having a direct child tag *paragraph*.
/book/@author	Any document with a top-level *book* tag having an attribute *author*.
/book//@author	Any document with a top-level *book* tag having a descendant tag at any level with the attribute *author*.
/book[@author contains("barnes") and @title = "the lemon table"]	Any document with a top-level *book* tag with an *author* attribute containing "barnes" (normalized) and a *title* attribute that only contains the words "the," "lemon" and "table" (normalized in that order).
/book[@author contains("barnes") and (@title contains("lemon") or @title contains("flaubert"))]	Any document with a top-level *book* tag with the specified *author* attribute and either of the two specified *title* attributes.
/book[@publishDate > xs:date("2000-01-01")]	Top level tag is *book* and *book* has an attribute *PublishDate* that is greater than the date of 2000-01-01.
/book[purchaseTime > xs:dateTime("2009-05-20T13:00:00")]	Top level tag is *book*. *book* has a direct child *purchaseTime* that is a DateTime expression greater than 2009-05-20T13:00:00.
/program[. contains("""hello, world.""")]	Any document with a top-level *program* tag containing both the tokens *hello* and *world* (normalized) in that order and in consecutive positions.

Table 250. Examples of valid XPath queries (continued)

Query	Description
/book[paragraph contains("foo")]//sentence	Any document with a top-level *book* tag with a direct child tag *paragraph* containing "foo" and, referring to the *book* tag, having a descendant tag *sentence* at any level.
/auto[@price < 30000.]	Any document with a top-level *auto* tag having an attribute *price* with a numeric value that is less than 30000.
//microbe[@size < 3.0e-06]	Any document containing a *microbe* tag at any level with a *size* attribute with a value that is less than .000003.

Text search language options

The text search language option specifies which language rules to use when performing a text search.

If QUERYLANGUAGE is not specified, the default is the language value of the text search index that is used when the CONTAINS or SCORE function is invoked. If the language value of the text search index is AUTO, the default value for QUERYLANGUAGE is en_US. The following table shows the valid language codes that may be used in the QUERYLANGUAGE option.

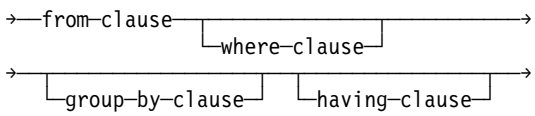
Language code	Language
ar_AA	Arabic
cs_CZ	Czech
da_DK	Danish
de_CH	German (Switzerland)
de_DE	German (Germany)
el_GR	Greek
en_AU	English (Australia)
en_GB	English (United Kingdom)
en_US	English (United States)
es_ES	Spanish (Spain)
fi_FI	Finnish
fr_CA	French (Canada)
fr_FR	French (France)
it_IT	Italian
ja_JP	Japanese
ko_KR	Korean
nb_NO	Norwegian Bokmal
nl_NL	Dutch
nn_NO	Norwegian Nynorsk
pl_PL	Polish
pt_BR	Portuguese (Brazil)
pt_PT	Portuguese (Portugal)
ru_RU	Russian
sv_SE	Swedish
zh_CN	Simplified Chinese
zh_TW	Traditional Chinese

Appendix H. Terminology differences

Some terminology used in the ANSI and ISO standards differs from the terminology used in this book and other product books.

The following table is a cross reference of the SQL 2003 Core standard terms to DB2 SQL

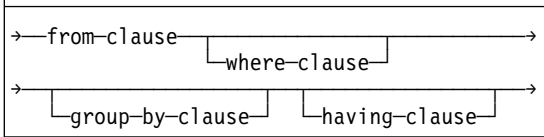
Table 251. ANSI/ISO term to DB2 SQL term cross-reference

ANSI/ISO Term	**DB2 SQL Term**
literal	constant
comparison predicate	basic predicate
comparison predicate subquery	subquery in a basic predicate
degree of table/cursor	number of items in a select list
grouped table	result table created by a group-by or having clause
grouped view	result view created by a group-by or having clause
grouping column	column in a group-by clause
outer reference	correlated reference
query expression	fullselect
query specification	subselect
result specification	result
set function	aggregate function
table expression	
target specification	host variable followed by an indicator variable
transaction	logical unit of work or unit of work
value expression	arithmetic expression

The following table is a cross reference of DB2 SQL terms to the SQL 2003 Core standard terms.

Terminology differences

Table 252. DB2 SQL term to ANSI/ISO term cross-reference

DB2 SQL Term	ANSI/ISO Term
aggregate function	set function
arithmetic expression	value expression
basic predicate	comparison predicate
column in a group-by clause	grouping column
correlated reference	outer reference
	table expression
fullselect	query expression
host variable followed by an indicator variable	target specification
logical unit of work or unit of work	transaction
interactive SQL	direct SQL
number of items in a select list	degree of table/cursor
result	result specification
result table created by a group-by or having clause	grouped table
result view created by a group-by or having clause	grouped view
subquery in a basic predicate	comparison predicate subquery
subselect	query specification
subselect or fullselect in parenthesis	query term

Appendix I. Reserved schema names and reserved words

This topic describes the restrictions of certain names used by the database manager. In some cases, names are reserved and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs though not prevented by the database manager.

Reserved schema names

This is the list of reserved schema names.

The following schema names are reserved:

- QSYS2
- SYSCAT
- SYSFUN
- SYSIBM
- SYSIBMADM
- SYSPROC
- SYSPUBLIC
- SYSSTAT
- SYSTEM

In addition, it is strongly recommended that schema names never begin with the Q prefix or SYS prefix, as Q and SYS are by convention used to indicate an area reserved by the system.

It is also recommended not to use SESSION as a schema name.

Reserved words

This is the list of currently reserved DB2 for i words.

Words may be added at any time. For a list of additional words that may become reserved in the future, see the IBM SQL and ANSI reserved words in the SQL Reference for Cross-Platform Development (http://www.ibm.com/developerworks/data/library/techarticle/0206sqlref/0206sqlref.html).

Table 253. SQL Reserved Words

ACCORDING	COLUMN	DECLARE	FIELDPROC
ACCTNG	COMMENT	DEFAULT	FILE
ACTION	COMMIT	DEFAULTS	FINAL
ACTIVATE	COMPACT	DEFER	FOR
ADD	COMPRESS	DEFINE	FOREIGN
ALIAS	CONCAT	DEFINITION	FREE
ALL	CONCURRENT	DELETE	FREEPAGE
ALLOCATE	CONDITION	DELETING	FROM
ALLOW	CONNECT	DENSERANK	FULL
ALTER	CONNECT_BY_ROOT	DENSE_RANK	FUNCTION
AND	CONNECTION	DESC	GBPCACHE
ANY	CONSTRAINT	DESCRIBE	GENERAL
APPEND	CONTAINS	DESCRIPTOR	GENERATED
APPLNAME	CONTENT	DETERMINISTIC	GET
ARRAY	CONTINUE	DIAGNOSTICS	GLOBAL
ARRAY_AGG	COPY	DISABLE	GO
AS	COUNT	DISALLOW	GOTO
ASC	COUNT_BIG	DISCONNECT	GRANT
ASENSITIVE	CREATE	DISTINCT	GRAPHIC
ASSOCIATE	CROSS	DO	GROUP
AT	CUBE	DOCUMENT	HANDLER
ATOMIC	CURRENT	DOUBLE	HASH
ATTRIBUTES	CURRENT_DATE	DROP	HASHED_VALUE
AUTHORIZATION	CURRENT_PATH	DYNAMIC	HAVING
BEFORE	CURRENT_SCHEMA	EACH	HINT
BEGIN	CURRENT_SERVER	ELSE	HOLD
BETWEEN	CURRENT_TIME	ELSEIF	HOUR
BINARY	CURRENT_TIMESTAMP	ENABLE	HOURS
BIND	CURRENT_TIMEZONE	ENCRYPTION	ID
BIT	CURRENT_USER	END	IDENTITY
BUFFERPOOL	CURSOR	ENDING	IF
BY	CYCLE	END-EXEC (COBOL only)	IGNORE
CACHE	DATA	ENFORCED	IMMEDIATE
CALL	DATABASE	ESCAPE	IMPLICITLY
CALLED	DATAPARTITIONNAME	EVERY	IN
CARDINALITY	DATAPARTITIONNUM	EXCEPT	INCLUDE
CASE	DATE	EXCEPTION	INCLUDING
CAST	DAY	EXCLUDING	INCLUSIVE
CCSID	DAYS	EXCLUSIVE	INCREMENT
CHAR	DBINFO	EXECUTE	INDEX
CHARACTER	DBPARTITIONNAME	EXISTS	INDEXBP
CHECK	DBPARTITIONNUM	EXIT	INDICATOR
CL	DB2GENERAL	EXTEND	INF
CLOSE	DB2GENRL	EXTERNAL	INFINITY
CLUSTER	DB2SQL	EXTRACT	INHERIT
COLLECT	DEACTIVATE	FENCED	INLINE
COLLECTION	DEALLOCATE	FETCH	INNER

Table 254. SQL Reserved Words (continued)

	INOUT	NATIONAL	PIECESIZE	SECOND
	INSENSITIVE	NCHAR	PIPE	SECONDS
	INSERT	NCLOB	PLAN	SECQTY
	INSERTING	NEW	POSITION	SELECT
	INTEGRITY	NEW_TABLE	PREPARE	SENSITIVE
	INTERSECT	NEXTVAL	PREVVAL	SEQUENCE
	INTO	NO	PRIMARY	SESSION
	IS	NOCACHE	PRIOR	SESSION_USER
	ISOLATION	NOCYCLE	PRIQTY	SET
	ITERATE	NODENAME	PRIVILEGES	SIGNAL
	JAVA	NODENUMBER	PROCEDURE	SIMPLE
	JOIN	NOMAXVALUE	PROGRAM	SKIP
	KEY	NOMINVALUE	PROGRAMID	SNAN
	LABEL	NONE	QUERY	SOME
	LANGUAGE	NOORDER	RANGE	SOURCE
	LATERAL	NORMALIZED	RANK	SPECIFIC
	LEAVE	NOT	RCDFMT	SQL
	LEFT	NULL	READ	SQLID
	LEVEL2	NULLS	READS	STACKED
	LIKE	NVARCHAR	RECOVERY	START
	LIMIT	OBID	REFERENCES	STARTING
	LINKTYPE	OF	REFERENCING	STATEMENT
	LOCAL	OFFSET	REFRESH	STATIC
	LOCALDATE	OLD	REGEXP_LIKE	STOGROUP
	LOCALTIME	OLD_TABLE	RELEASE	SUBSTRING
	LOCALTIMESTAMP	ON	RENAME	SUMMARY
	LOCATION	OPEN	REPEAT	SYNONYM
	LOCATOR	OPTIMIZE	RESET	SYSTEM_USER
	LOCK	OPTION	RESIGNAL	TABLE
	LOCKSIZE	OR	RESTART	TABLESPACE
	LOG	ORDER	RESULT	TABLESPACES
	LOGGED	ORDINALITY	RESULT_SET_LOCATOR	THEN
	LONG	ORGANIZE	RETURN	THREADSAFE
	LOOP	OUT	RETURNS	TIME
	MAINTAINED	OUTER	REVOKE	TIMESTAMP
	MATCHED	OVER	RID	TO
	MATERIALIZED	OVERLAY	RIGHT	TRANSACTION
	MAXVALUE	OVERRIDING	ROLLBACK	TRIGGER
	MERGE	PACKAGE	ROLLUP	TRIM
	MICROSECOND	PADDED	ROUTINE	TRIM_ARRAY
	MICROSECONDS	PAGE	ROW	TYPE
	MINPCTUSED	PAGESIZE	ROWNUMBER	UNDO
	MINUTE	PARAMETER	ROW_NUMBER	UNION
	MINUTES	PART	ROWS	UNIQUE
	MINVALUE	PARTITION	RRN	UNIT
	MIXED	PARTITIONED	RUN	UNNEST
	MODE	PARTITIONING	SAVEPOINT	UNTIL
	MODIFIES	PARTITIONS	SBCS	UPDATE
	MONTH	PASSING	SCHEMA	UPDATING
	MONTHS	PASSWORD	SCRATCHPAD	URI
	NAMESPACE	PATH	SCROLL	USAGE
	NAN	PCTFREE	SEARCH	USE

Table 255. SQL Reserved Words (continued)

USER	WAIT	XMLATTRIBUTES	XMLROW
USERID	WHEN	XMLCAST	XMLSERIALIZE
USING	WHENEVER	XMLCOMMENT	XMLTABLE
VALUE	WHERE	XMLCONCAT	XMLTEXT
VALUES	WHILE	XMLDOCUMENT	XMLVALIDATE
VARIABLE	WITH	XMLELEMENT	XSLTRANSFORM
VARIANT	WITHOUT	XMLFOREST	XSROBJECT
VCAT	WRAPPED	XMLGROUP	YEAR
VERSION	WRITE	XMLNAMESPACES	YEARS
VIEW	WRKSTNNAME	XMLPARSE	YES
VOLATILE	XMLAGG	XMLPI	

Appendix J. Related information

The publications listed here provide additional information about topics described or referred to in this guide.

These manuals are listed with their full titles and order numbers. When these manuals are referred to in this guide, a shortened version of the title is used.

- Backup and recovery

  The manual contains information about planning a backup and recovery strategy, the different types of media available to save and restore procedures, and disk recovery procedures. It also describes how to install the system again from backup.

- ILE COBOL Programmer's Guide 

  This guide provides information needed to design, write, test, and maintain COBOL programs on the IBM i products.

- ILE RPG Programmer's Guide 

  This guide provides information you need to design, write, test, and maintain ILE RPG programs on the IBM i products.

- REXX/400 Programmer's Guide 

  This guide provides information you need to design, write, test, and maintain REXX/400 programs on the IBM i products.

- CL programming

  This guide provides a wide-ranging discussion of the programming topics, including a general discussion of objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.

- Database file management

  This book provides information about using files in application programs.

- Database programming

  This book provides a detailed description of the IBM i database organization, including information about how to create, describe, and update database files on the system.

- Distributed database programming

  Provides information about preparing and managing an IBM i product in a distributed relational database using the Distributed Relational Database Architecture (DRDA). Describes planning, setting up, programming, administering, and operating a distributed relational database on more than one IBM i product in a like-system environment.

- Security reference

  This guide provides information about system security concepts, planning for security, and setting up security on the system. It also gives information about protecting the system and data from being used by people who do not have the

proper authorization, protecting the data from intentional or unintentional damage or destruction, keeping security up-to-date, and setting up security on the system.

- SQL programming

  This book provides an overview of how to design, write, run, and test DB2 for i statements. It also describes interactive Structured Query Language (SQL).

- SQL XML programming

  This book provides information about using the XML data type with SQL. It includes examples of using functions and procedures to generate XML and to retrieve all or part of an XML document.

- Embedded SQL programming

  This book provides examples of how to write SQL statements in ILE C, ILE C++, COBOL, ILE COBOL, RPG, ILE RPG, REXX, and PL/I programs.

- Database performance and query optimization

  This book provides information about optimizing the performance of your queries using available tools and techniques.

- IDDU Use 

  This book describes how to use IBM i interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system.

- SQL call level interfaces (ODBC)

  This book describes how to use X/Open SQL Call Level Interface to access SQL functions directly through procedure calls to a service program provided by DB2 for i.

- System i Access category in the IBM i Information Center

  This information describes how to set up and run ODBC applications on a client using System i Access ODBC. Included in this document are chapters on performance, examples, and configuring specific applications to run with System i Access ODBC.

- IBM Toolbox for Java

  This book describes how to set up and run JDBC applications on a client using IBM Toolbox for Java. Included in this document are chapters on performance, examples, and configuring specific applications to run with IBM Toolbox for Java.

- IBM Developer Kit for Java

  This information provides the details you need to design, write, test, and maintain JAVA programs on the IBM i product. The book also contains information about the IBM Developer Kit for Java JDBC driver.

- DB2 Multisystem

  This book describes the fundamental concepts of distributed relational database files, nodegroups, and partitioning. The book provides the information you need to create and use database files that are partitioned across multiple systems. Information is provided on how to configure the systems, how to create the files, and how the files can be used in applications.

Appendix K. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This DB2 for i SQL Reference publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Printed in USA