SQL Reference for Cross-Platform Development - Version 5

Contents

Chapter 1. About this book

This book defines IBM® DB2® Structured Query Language (DB2 SQL). It describes the rules and limits for preparing portable programs. This book is a reference rather than a tutorial and assumes a familiarity with SQL programming concepts.

Who should read this book

This book is intended for programmers who want to write portable applications using SQL that is common to the DB2 relational database products and the SQL 2011 Core standard.[1] DB2 SQL is consistent with the SQL 2011 Core standard. DB2 SQL also provides functional extensions to the SQL 2011 Core standard. For example, many of the scalar functions defined in this book are extensions to the SQL 2011 Core standard.

1. In this book, the term "SQL 2011 Core standard" is used to describe the ANSI/ISO Core Level SQL standard of 2011 and related industry standards. See "Related documentation" on page 6 for a list of documentation that describes these industry standards.

How to use this book

This book defines the DB2 SQL language elements that are common to the IBM DB2 Family of relational database products across the following environments:

Environment	IBM Relational Database Product	Short Name
z/OS®	DB2 11 for z/OS	DB2 for z/OS
i operating system	DB2 for IBM i Version 7.2	DB2 for i
Linux UNIX Windows	DB2 10.5 for the Linux, UNIX, and Windows Platforms Version 10.5	DB2 for LUW

The DB2 relational database products have product books that also describe product-specific elements and explain how to prepare and run a program in a particular environment. The information in this book is a subset of the information in the product books, and the rules and limits described in this book might not apply to all products. The limits in this book are those required to assist program portability across the applicable IBM environments. See "Related documentation" on page 6 for a list of the product books needed in addition to this book.

The SQL described in this book assumes that default environment options, including:

- precompile options
- bind options
- registry variables

are set to default values, unless specifically mentioned.

Since each DB2 product does not ship on the same schedule, at any point in time there will naturally be some elements that are only available on a subset of the DB2 products. Some elements might be implemented by all products, but differ slightly in their semantics (how they behave when the program is run). In many cases, these semantic difference are the result of the underlying operating system support. These conditions are identified in this book as shown in the next paragraph.

The DB2 SQL definition is described in this book. If the implementation of an element in a product differs from the DB2 SQL definition in its syntax or semantics, the difference is highlighted with a symbol in the left margin as is this sentence.

Assumptions relating to examples of SQL statements

The examples of SQL statements shown in this guide are based on the sample tables in Chapter 20, "Sample tables," on page 1145 and assume the following:

- SQL keywords are highlighted.
- Table names used in the examples are the sample tables.

Code disclaimer information

This document contains programming examples.

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

All sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All programs contained herein are provided to you "AS IS" without any warranties of any kind. The implied warranties of non-infringement, merchantability and fitness for a particular purpose are expressly disclaimed.

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.

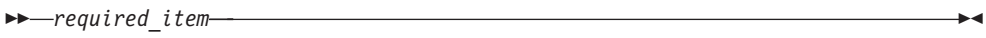
  The ►►— symbol indicates the beginning of the syntax diagram.

  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).



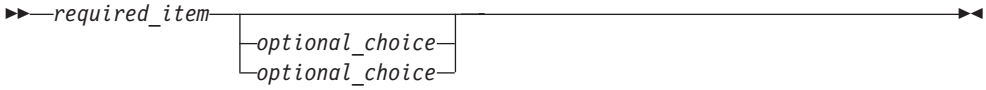
- Optional items appear below the main path.

  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.

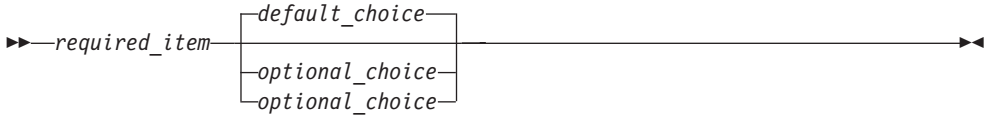
- If more than one item can be chosen, they appear vertically, in a stack.

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

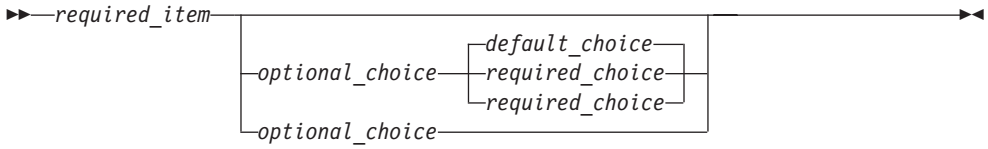
  If choosing one of the items is optional, the entire stack appears below the main path.



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

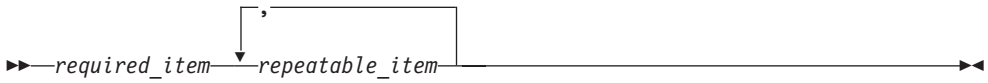


If an optional item has a default when it is not specified, the default appears above the main path.



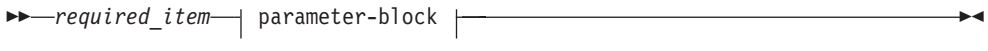
- An arrow returning to the left, above the main line, indicates an item that can be repeated.

If the repeat arrow contains a comma, repeated items must be separated with a comma.

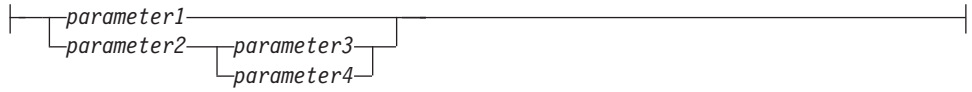


A repeat arrow above a stack indicates that the items in the stack can be repeated.

- 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, they must be entered as part of the syntax.
- The syntax diagrams only contain the preferred or standard keywords. If non-standard 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 manual

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

Highlighting conventions

The following conventions are used in this book.

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[2]:

Convention	Representation
SO	"shift-out" control character (X'0E'), used only for EBCDIC data
SI	"shift-in" control character (X'0F'), used only for EBCDIC data
sbcs-string	SBCS string of zero or more single-byte characters
dbcs-string	DBCS string of zero or more double-byte characters
'	DBCS apostrophe
G	DBCS uppercase G

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 on IBM's Accessibility support visit the Accessibility Center at http://www.ibm.com/able/.

SQL accessibility support falls in two main categories.

- The DB2 products provide Windows graphical user interfaces to DB2 databases. For information about the Accessibility features supported in Windows graphical user interfaces, see Accessibility in the Windows Help Index.

2. Hexadecimal values are for EBCDIC characters.

- 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 on the IBM Home Page Reader and other tools, visit the Accessibility Center.

The IBM Home Page Reader can be used to access all descriptive text in this book.

Related documentation

The following documentation for DB2 is available on the internet at:

http://www.ibm.com/software/data/db2/

DB2 for z/OS

- *DB2 for z/OS SQL Reference*
- *DB2 for z/OS Application Programming and SQL Guide*
- *DB2 for z/OS Application Programming Guide and Reference for Java*
- *DB2 for z/OS in the IBM Knowledge Center*, at http://www.ibm.com/support/knowledgecenter/SSEPEK/db2z_prodhome.html

DB2 for IBM i

- *DB2 for i SQL Reference*
- *DB2 for i SQL Programming*
- *DB2 for i Embedded SQL Programming*
- *DB2 for IBM i in the IBM Knowledge Center*, at http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/
- *IBM Developer Kit for Java in the IBM Knowledge Center*

DB2 10.5 for the Linux, UNIX, and Windows Platforms

- *IBM DB2 for Linux, UNIX, and Windows SQL Reference*, Volumes 1 and 2
- *IBM DB2 for Linux, UNIX, and Windows Getting Started with Database Application Development*
- *IBM DB2 for Linux, UNIX, and Windows Developing Embedded SQL Applications*
- *IBM DB2 for Linux, UNIX, and Windows Developing User-defined Routines (SQL and External)*
- *IBM DB2 for Linux, UNIX, and Windows Developing Java Applications*
- *DB2 for Linux UNIX and Windows in the IBM Knowledge Center*, at http://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.kc.doc/welcome.html

Distributed relational database architecture

- *Application Programming Guide*, SC26-4773
- *Connectivity Guide*, SC26-4783
- *Open Group Publications: DRDA Vol. 1: Distributed Relational Database Architecture (DRDA)*, at http://www.opengroup.org/publications/catalog/c066.htm.
- *Open Group Publications: DRDA Vol. 2: Formatted Data Object Content Architecture (FD:OCA)*, at http://www.opengroup.org/publications/catalog/c067.htm.
- *Open Group Publications: DRDA Vol. 3: Distributed Data Management (DDM) Architecture*, at http://www.opengroup.org/publications/catalog/c068.htm.

Character data representation architecture

- *Character Data Representation Architecture Reference and Registry*, SC09-2190

Unicode standard

- *The Unicode Standard*, at http://www.unicode.org.

Industry 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 9075-13:2008, Information technology - Database languages - SQL - Part 13: Java Routines and Types (SQL/JRT)*
- *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 9075-13:2011, Information technology - Database languages - SQL - Part 13: Java Routines and Types (SQL/JRT)*
- *INCITS/ISO/IEC 9075-14:2011, Information technology - Database languages - SQL - Part 14: XML-Related Specifications (SQL/XML)*

What's new for this book

The major new features covered in this book include:

- ARRAY element specification and Array constructor
- TIMESTAMP precision
- New built-in functions:
  - *Aggregate Functions*: ARRAY_AGG and GROUPING
  - *Miscellaneous Scalar Functions*: CARDINALITY, MAX_CARDINALITY, TRIM_ARRAY, and VERIFY_GROUP_FOR_USER
  - *Miscellaneous Scalar Functions*: LPAD and RPAD
  - *Table Functions*: XMLTABLE
- Miscellaneous Scalar function enhancements
- *Built-in global variables*: CLIENT_IPADDR, PACKAGE_NAME, PACKAGE_SCHEMA, and PACKAGE_VERSION
- GROUP BY referencing super groups (CUBE & ROLLUP) and grouping sets
- Row and column access control for tables
  - ALTER FUNCTION (SQL)
  - ALTER MASK
  - ALTER PERMISSION
  - ALTER TRIGGER
  - CREATE MASK
  - CREATE PERMISSION
- ALTER TABLE DROP COLUMN RESTRICT
- COMMIT ON RETURN and AUTONOMOUS procedures
- CREATE FUNCTION (SQL Scalar) allows more *SQL-control statements* in an *SQL-routine-body*
- CREATE INDEX on expressions
- CREATE TYPE (Array)
- CREATE VARIABLE
- GRANT and REVOKE (Variable privileges)
- TRUNCATE
- Multiple assignments on a SET statement
- Arrays in *compound-statement*

Chapter 2. 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, types, functions, procedures, sequences, and packages. Any number of relational databases can be created on a given physical machine.

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 one of the supported host languages; COBOL, C (C also covers C++ in this documentation, unless otherwise mentioned explicitly) 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 invoke 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. The exact steps required are product-specific.

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 a dynamic statement is a character 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. For DB2 for z/OS it only persists for the duration of the transaction.

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

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*.

For example, the following facilities provide interactive capabilities:

- SPUFI for DB2 for z/OS
- System i® Navigator, Interactive SQL, or Query Manager for DB2 for i.
- Command Line Processor or Command Center for DB2 for LUW.

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 to access SQL functions directly through a call interface. 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.

For a complete description of all the available functions, see the product books.

Java database connectivity and embedded SQL for Java programs

DB2 provides 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.

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, see the product books.

Schemas

The objects in a relational database are organized into sets called schemas. A schema provides a logical classification of objects in the database. The *schema name* is used as the qualifier of SQL object names such as tables, views, indexes, and triggers.

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.

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.

A *base table* is created with the CREATE TABLE statement and is used to hold persistent user data. For more information about creating tables, see "CREATE TABLE" on page 688.

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 fullselect. The fullselect 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 688.

A *partitioned table* is a table whose data is contained in one or more data partitions. 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.

A *result table* is a set of rows that the database manager selects or generates from a query. For information on queries, see Chapter 6, "Queries," on page 463.

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 columns that are identified as such in the description of an index, a unique constraint, or a referential constraint. The same column can be part of more than one key.

A *composite key* is an ordered set of two or more columns of the same base table. The ordering of the columns 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 using the PRIMARY KEY clause of the CREATE TABLE or ALTER TABLE statement. A base table cannot have more than one primary key and the columns of the key must be defined as NOT NULL. 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 UNIQUE key. The columns of a UNIQUE key must be defined as NOT NULL.

The unique index that is used to enforce a unique constraint may be implicitly created when the unique constraint is defined. Alternatively, it can be defined by using the CREATE UNIQUE INDEX statement. In DB2 for z/OS, indexes are only implicitly created if the CREATE TABLE or ALTER TABLE statement is processed by the schema processor or the table space is implicitly created.

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.

For more information on defining unique constraints, see "ALTER TABLE" on page 547 or "CREATE TABLE" on page 688.

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 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 base 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 that a nonnull update 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 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, or SET NULL. 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.

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 descendents 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 base tables and may affect rows of these tables:

- If 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 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 547 or “CREATE TABLE” on page 688.

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 on search conditions, see "Search conditions" on page 178.

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 on defining check constraints, see "ALTER TABLE" on page 547 or "CREATE TABLE" on page 688.

Indexes

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

Indexes are used by the database manager to:

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

An *index* is created with the CREATE INDEX statement. For more information about creating indexes, see "CREATE INDEX" on page 653.

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.

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 the 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, the 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 718.

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.

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.

- 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.

Views

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

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 constraint 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* is created with the CREATE VIEW statement. For more information about creating views, see "CREATE VIEW" on page 743.

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 734 and "User-defined types" on page 70.

An *array type* is a user-defined data type that consists of an ordered set of elements of a single built-in data type. Elements can be accessed and modified by their index position. An array type is created with an SQL CREATE TYPE (array) statement. An array type can be used as a parameter of an procedure or scalar

function and as a variable in an SQL procedure or SQL scalar function. For more information, see "CREATE TYPE (array)" on page 730

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. 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 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* is created with the CREATE ALIAS statement. For more information about creating aliases, see "CREATE ALIAS" on page 605.

Packages and access plans

A *package* is an object that contains control structures used to execute SQL statements. Packages are produced during program preparation. The control structures can be thought of as the bound or operational form of SQL statements. 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 other product-specific objects 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 776). This means that any packages, procedures, functions, triggers, and any product-specific objects containing control structures referencing the dropped object are invalidated.

In some cases, an invalidated *access plan* may be automatically 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.

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 123.

Functions are classified as either SQL functions or external functions. SQL functions are written using an SQL RETURN statement, which is part of the 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 606.

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 666.

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. SQL can also easily 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 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 then waits for a single reply that indicates whether the CALL statement executed successfully or not and optionally returns results.

The following two figures illustrate the way procedures can be used in a distributed application to eliminate some of the remote requests. Figure 1 on page 20 shows a program making many remote requests.

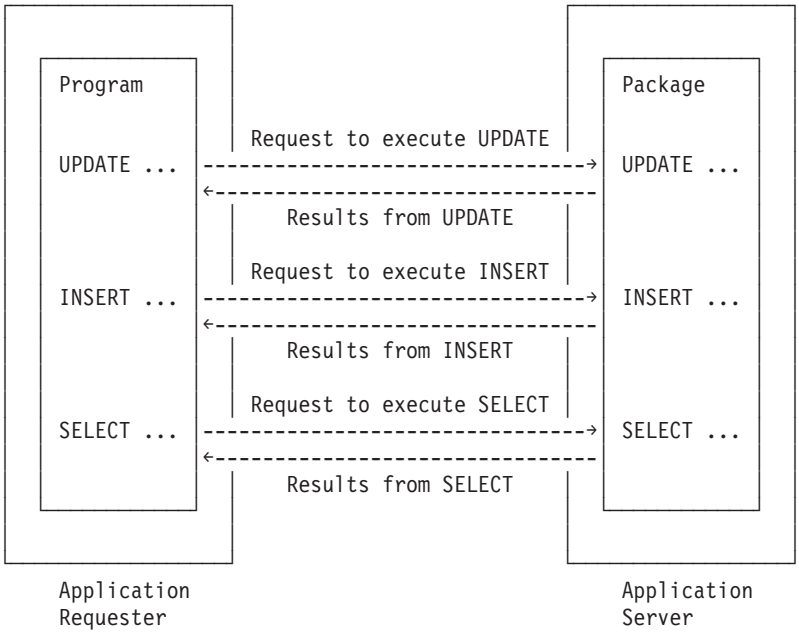


Figure 1. Application without remote procedure

Figure 2 shows how a call to a remote package can reduce the number of remote requests.

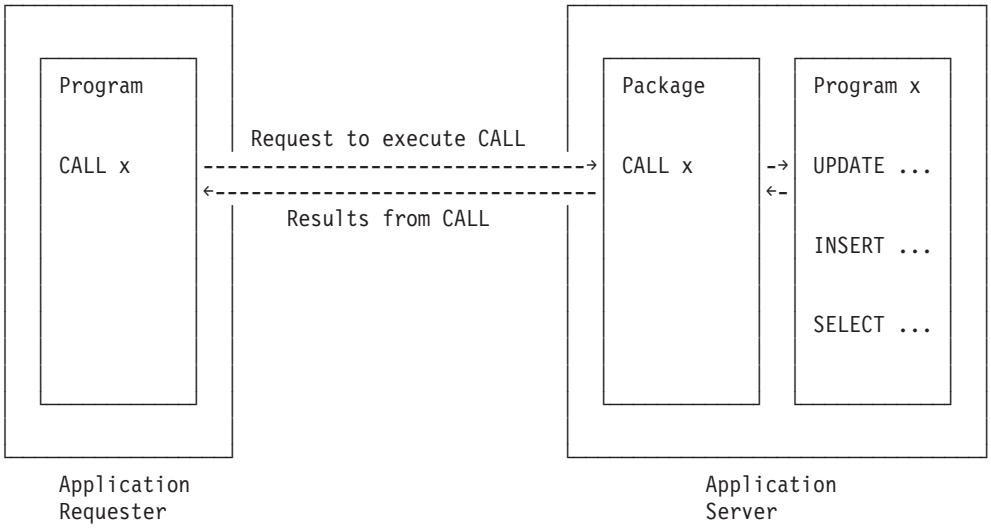


Figure 2. Application with remote procedure

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 "CREATE SEQUENCE" on page 682 and "Sequence reference" on page 158.

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.

The two forms of authorization are administrative authority and privileges.

Administrative authority

The holder of an administrative authority is charged with the task of managing the relational database and is responsible for the safety and integrity of the data.

database administrator authority
The *database administrator authority* provides a user with the ability to create and manage all objects in a relational database. Those with *database administrator authority* implicitly have all privileges on all objects in the relational database.

For further information about *database administrator authority*, see the product references.

security administrator authority

The *security administrator authority* provides a user the ability to manage security in a relational database. The *security administrator authority* possesses the ability to grant and revoke all relational database privileges and authorities. Those with *security administrator authority* also manage security policies by enforcing row and column access control for tables that contain sensitive data.

The *security administrator authority* has no inherent privilege to access data stored in tables.

For further information about *security administrator authority*, see the product references.

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).

- In DB2 for z/OS and DB2 for LUW, the set consists of all users (including future users), including those with privately granted privileges on the table or view.
- In DB2 for i, 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. Thus, if HERNANDZ needs both the SELECT and UPDATE privileges, both privileges must be granted.

The REVOKE statement can be used to REVOKE previously granted *privileges*.

- In DB2 for z/OS a revoke of a privilege from an authorization ID only revokes the privilege granted by a specific authorization ID. For example, assume that the SELECT has been granted to CHRIS by CLAIRE and also by BOBBY. If CLAIRE revokes the SELECT privilege from CHRIS, CHRIS still has the SELECT privilege that was granted by BOBBY.

  Revoking a privilege from an authorization ID will also revoke that same privilege from all other authorization IDs that were granted the privilege by that authorization ID. For example, assume CLAIRE grants SELECT WITH GRANT OPTION to RICK, and RICK then grants SELECT to BOBBY and CHRIS. If CLAIRE revokes the SELECT privilege from RICK, the SELECT privilege is also revoked from both BOBBY and CHRIS.
- In DB2 for i, and DB2 for LUW, 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. For example, assume CLAIRE grants SELECT WITH GRANT OPTION to RICK, and RICK then grants SELECT to BOBBY and CHRIS. If CLAIRE revokes the SELECT privilege from RICK, BOBBY and CHRIS still retain the SELECT privilege.

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
- G it has a product-specific privilege.

When an object is created, one authorization ID is assigned *ownership* of the object. Ownership means the user is authorized to reference the object in any applicable SQL statement. The privileges on the object can be granted by the owner, and cannot be revoked from the owner.

When an object is created, the authorization ID of the statement is the *owner* of an object if:

- the object name in the CREATE statement is not qualified, or
- the explicitly specified schema name is the same as the authorization ID of the statement.

G Otherwise, the owner of the object is product-specific and the privileges held by
G the authorization ID of the statement must include database administrator
G authority.

Row permissions and column masks

A *row permission* expresses an access control rule for each row of a specific table. A row permission is in the form of a search condition that describes which rows users may access. Row permissions are applied after table privileges (like SELECT or INSERT) are checked.

A *column mask* expresses an access control rule for a specific column in a table. A column mask is in the form of a CASE expression that describes which column values users may access. Column masks are applied after table privileges (like SELECT or INSERT) are checked.

Row permissions and column masks can be created, changed, and dropped only by those with security administrator authority by using the CREATE MASK, CREATE PERMISSION, ALTER MASK, ALTER PERMISSION, and DROP statements. The definition of a permission or a mask can reference other objects. Those with security administrator authority do not need additional privileges to reference those objects, such as SELECT privilege to retrieve from a table or EXECUTE privilege to invoke a user-defined function, in the definition of the row permission or column mask. Multiple row permissions and column masks can be created for a table. Only one column mask can be created for each column in a table. A row permission or a column mask can be created before row or column access control is enforced for a table. The definition of the row permission and the column mask is stored in the catalog. However, the permission and the mask do not take effect until the ALTER TABLE statement with the ACTIVATE ROW ACCESS CONTROL clause is used to enforce row access control or the ACTIVATE COLUMN ACCESS CONTROL clause is used to enforce column access control on the table.

When an ALTER TABLE statement is used to explicitly activate row access control for a table, a default row permission is implicitly created for the table which prevents all access to the table. After row access controls have been activated for a table, if the table is referenced implicitly or explicitly in a data change statement and if multiple row permissions are defined for the table, a row access control

search condition is derived by using the logical OR operator with the search condition of each defined row permission.

When an ALTER TABLE statement is used to explicitly activate column access control for a table, access to the table is not restricted. However, if the table is referenced in an SQL statement, all column masks that have been created for the table are applied to mask the column values that are referenced in the output of the queries or to determine the column values that are used in the data change statements.

The authorization ID for the SQL statement that references a table with row permissions or column masks does not need authority to reference objects that are specified in the definitions of those row permissions or column masks.

Catalog

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

G For further information about the catalog, see the product books.

Application processes, concurrency, and recovery

All SQL programs execute as part of an *application process*. An application process involves the execution of one or more programs, and is the unit to which the database manager allocates resources and locks. Different application processes may involve the execution of different programs, or different executions of the
G same program. The means of starting and ending an application process are
G dependent on the environment.

Locking, commit, and rollback

More than one application process may request access to the same data at the same time. *Locking* is used to maintain data integrity under such conditions, by preventing, for example, two application processes from updating the same row of data simultaneously.

G The locking facilities of the database managers are similar but not identical. One of the common properties is that each of the database managers can acquire locks in order to prevent uncommitted changes made by one application process from being perceived by any other. The database manager will release all locks it has acquired on behalf of an application process when that process ends, but an application process itself can also explicitly request that locks be released sooner. This operation is called *commit*.

G In DB2 for z/OS and DB2 for i, a lock that protects the current row of a cursor
G from updates or deletes by concurrent application processes also protects the row

from Positioned UPDATEs and Positioned DELETEs that reference another cursor of the same application process. [3] In DB2 for LUW this protection does not apply.

Unit of work

Like the locking facilities, the recovery facilities of the database managers are similar but not identical. One common property is that each of the database managers provides a means of *backing out* uncommitted changes made by an application process. This might be necessary in the event of a failure on the part of an application process, or in a *deadlock* situation. An application process itself, however, can explicitly request that its database changes be backed out. This operation is called *rollback*.

A *unit of work* (also called a *transaction*, *logical unit of work*, or *unit of recovery*) is a recoverable sequence of operations within an application process. At any time, an application process has at most a single unit of work, but the life of an application process may involve many units of work as a result of commit or rollback operations.

Note: In addition to relational databases, the environment in which an SQL program executes may also include other types of recoverable resources. If this is the case, the scope and acceptability of the SQL COMMIT and ROLLBACK statements depend on the environment.

A unit of work is started when the first SQL statement in an application process or the first SQL statement after a commit or rollback is executed. A unit of work is ended by a commit operation, a rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes made within the unit of work it ends. While these changes remain uncommitted, other application processes are unable to perceive them and they can be backed out. [4] Once committed, these database changes are accessible by other application processes and can no longer be backed out by a rollback.

The start and end of a unit of work define points of consistency within an application process. 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 have been added to the second account is consistency reestablished. When both steps are complete, the commit operation can be used to end the unit of work, thereby making the changes available to other application processes.

3. In DB2 for i, Searched UPDATEs and Searched DELETEs are also included.

4. Except for isolation level uncommitted read, described in "Uncommitted read" on page 30.

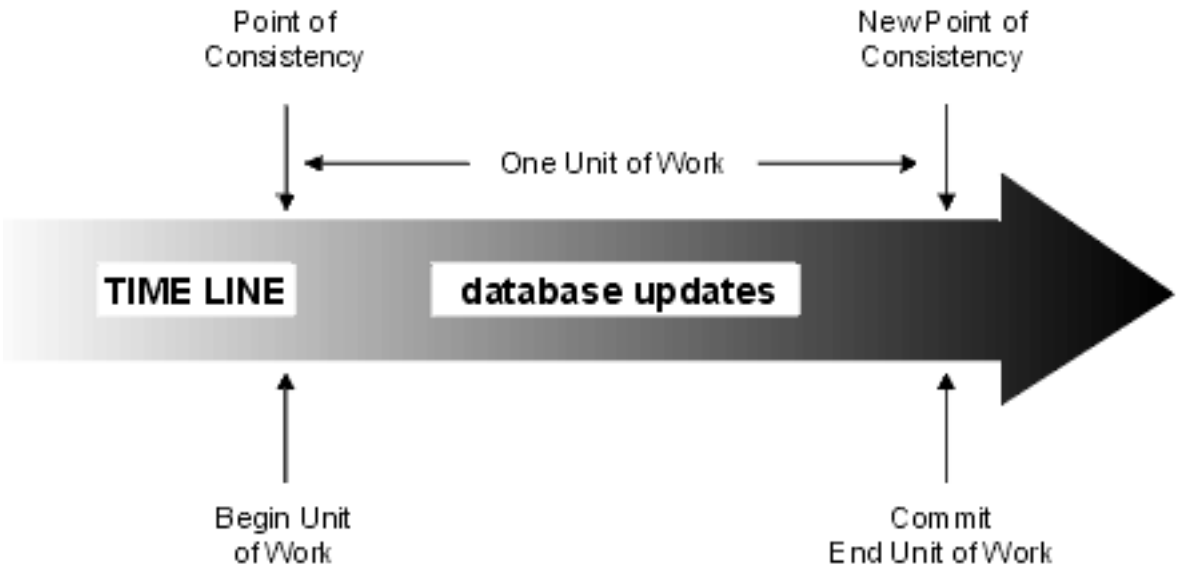


Figure 3. Unit of work with a commit operation

If a failure occurs before the unit of work ends, the database manager will back out uncommitted changes to restore the data consistency that existed when the unit of work was started.

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, the database manager backs out uncommitted changes to restore the data consistency that existed when the unit of work was initiated. That is, the database manager undoes the work, as shown in the diagram below:

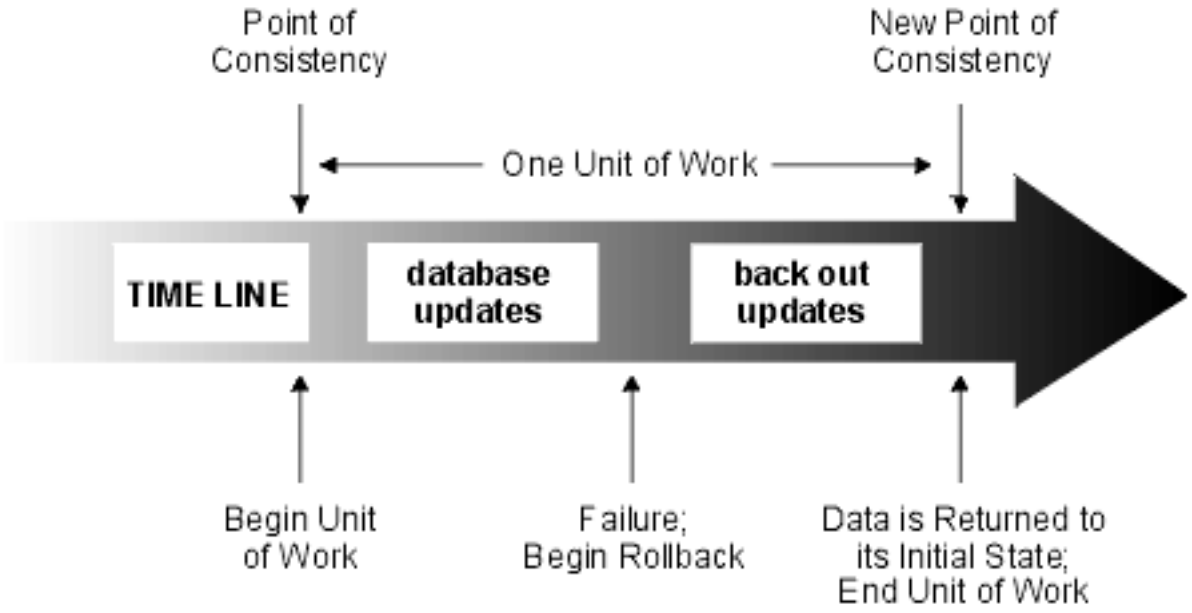


Figure 4. Rolling back changes from a unit of work

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.

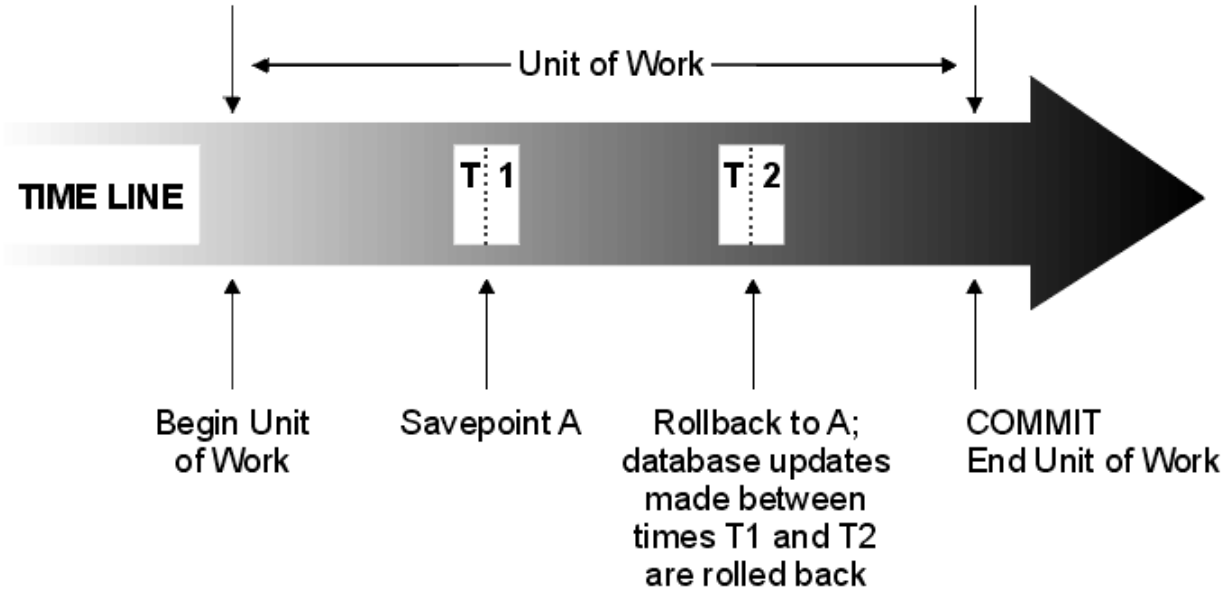


Figure 5. Unit of work with a ROLLBACK statement and a SAVEPOINT statement

Isolation level

The *isolation level* used during the execution of SQL statement determines the degree to which the application process is isolated from concurrently executing application processes. Thus, when application process P executes an SQL statement, the isolation level determines:

- The degree to which rows retrieved by P are available to other concurrently executing application processes.
- The degree to which database changes made by concurrently executing application processes 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*.

G Each product provides a product-specific means of explicitly specifying a default isolation level:

- For static SQL statements, the *default isolation level* is the isolation level specified when the containing package, procedure, function, or trigger was created.
- For dynamic SQL statements, the *default isolation level* is isolation level specified for the application process.

Products support these isolation levels by automatically locking the appropriate data. Depending on the type of lock, this limits or prevents access to the data by concurrent application processes. Each database manager supports at least two types of locks:

Share Limits concurrent application processes to read-only operations on the data.

Exclusive

Prevents concurrent application processes from accessing the data in any way except for application processes with an isolation level of *uncommitted read*, which can read but not modify the data. (See "Uncommitted read" on page 30.)

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 application process is guaranteed at least the minimum requested lock level.

Regardless of the isolation level, every database manager places exclusive locks on every row that is inserted, updated, or deleted. Thus, all isolation levels ensure that any row that is changed during a unit of work is not changed by any other application processes until the unit of work is complete. The isolation levels are:

Repeatable read

The Repeatable Read (RR) isolation level ensures that:

- Any row read during a unit of work is not changed by other application processes until the unit of work is complete. [5]
- Any row changed by another application process cannot be read until it is committed by that application process.

In addition to any exclusive locks, an application process running at level RR acquires at least share locks on all the rows it reads. Furthermore, the locking is performed so that the application process is completely isolated from the effects of concurrent application processes.

In the SQL 2011 Core standard, Repeatable Read is called Serializable.

Read stability

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

- Any row read during a unit of work is not changed by other application processes until the unit of work is complete. [5]
- Any row changed by another application process cannot be read until it is committed by that application process.

Unlike RR, RS does not completely isolate the application process from the effects of concurrent application processes. At level RS, application processes that issue the same query more than once in the same unit of work might see additional rows. These additional rows are called *phantom rows*.

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

1. Application process P1 reads the set of rows *n* that satisfy some search condition.
2. Application process 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.

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.

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

In the SQL 2011 Core standard, Read Stability is called Repeatable Read.

Cursor stability

Like levels RR and RS, the Cursor Stability (CS) isolation level ensures that any row changed by another application process cannot be read until it is committed by that application process. Unlike RR and RS, level CS only ensures that the current row of every updatable cursor is not changed by other application processes. Thus, the rows read during a unit of work can be changed by other application processes. In addition to any exclusive locks, an application process running at level CS has at least a share lock for the current row of every one of its open cursors.

In the SQL 2011 Core standard, Cursor Stability is called Read Committed.

Uncommitted read

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

- Any row read during the unit of work to be changed by other application processes.
- Any row changed by another application process to be read even if the change has not been committed by that application process.

G For other operations, the rules of level CS apply. In DB2 for z/OS, UR is escalated
G to CS for a subquery used in a DELETE or UPDATE statement, or for a subselect
G used in an INSERT statement.

In the SQL 2011 Core standard, Uncommitted Read is called Read Uncommitted.

Comparison of isolation levels

The following table summarizes information about isolation levels.

	UR	CS	RS	RR
Can the application see uncommitted changes made by other application processes?	Yes	No	No	No
Can the application update uncommitted changes made by other application processes?	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	No
Can “updated” rows be updated by other application processes?	No	No	No	No
Can “updated” rows be read by other application processes that are running at an isolation level other than UR?	No	No	No	No
Can “updated” rows be read by other application processes that are running at the UR isolation level?	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	No	No
Can "accessed" rows be read by other application processes?	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	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, 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

Storage structures (spaces for tables and indexes for example) differ between each DB2 relational database product. For detailed information about storage structures, see the product references.

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*. [6]

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

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

6. 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.

- 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 character 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 nonaccented letters A through Z
- 26 nonaccented letters a through z
- digits 0 through 9
- . , : ; ? ( ) ' " / - _ & + % * = < >

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 establishes 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 [7]
- Double-byte EBCDIC
- Mixed single- and double-byte ASCII
- Unicode (UTF-8, UTF-16, and UCS-2 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,

7. The term ASCII is used throughout this book to refer to several encodings such as IBM-PC, Windows, ISO 8, or ISO 7 data.

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, an 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 supplementary characters 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.

  UCS-2 is identical to UTF-16 except that UTF-16 also supports supplementary characters through the use of surrogates. Processing UCS-2 data is faster than UTF-16 data because each character in UCS-2 is assumed to be exactly 16-bits long[8]
- 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, the character 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 as equal, it is usually not a good idea to store both forms of the characters in the database. *Normalization* is a process that replaces characters that contain 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 the character é) will be converted to X'00E9' (the normalized representation of the character é).

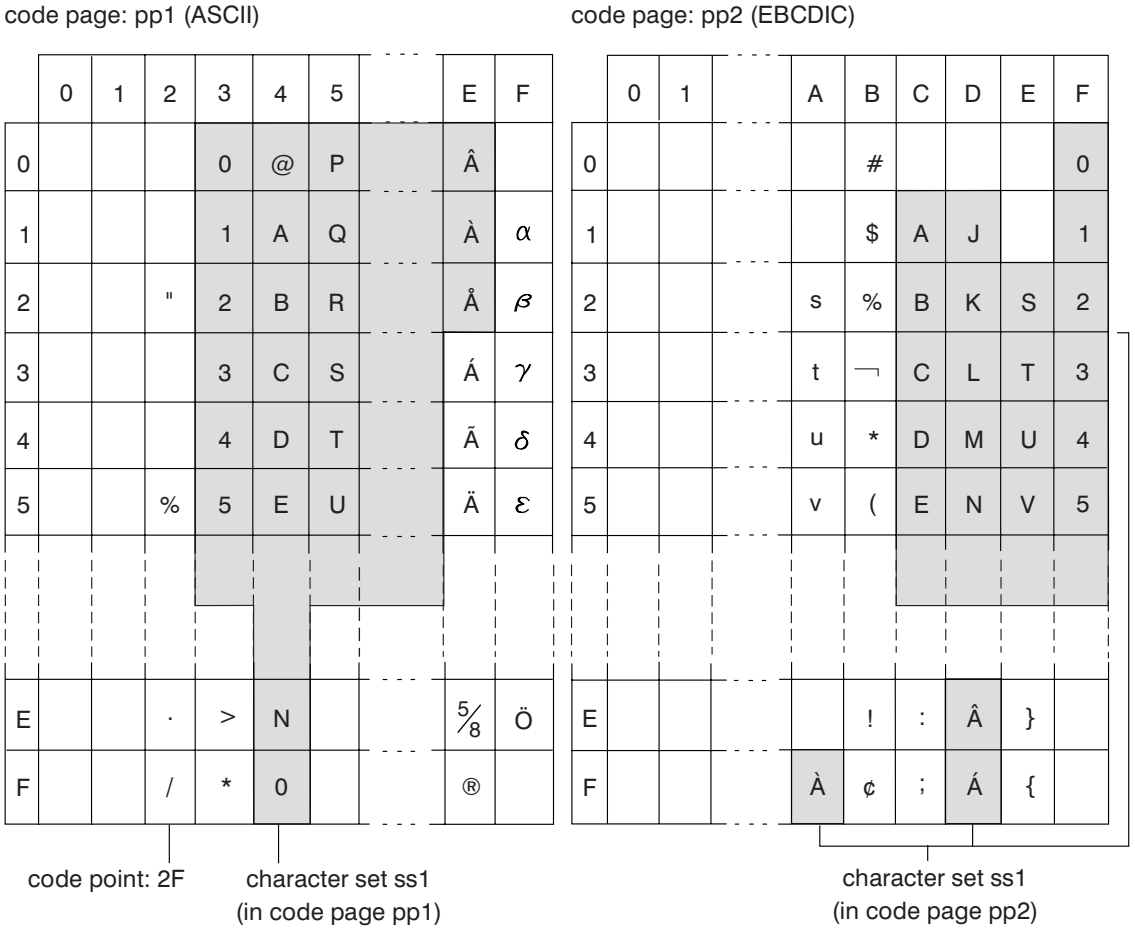
  Both UTF-8 and UTF-16 can contain 4 byte characters called *surrogates*. Surrogates are 4 byte sequences that can address one million more characters than would be available in a 2 byte character set.

G In DB2 for LUW, the CCSID values of both 13488 and 1200 are used to
G specify UCS-2 data.

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

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.



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 (that is a mixture of single-byte characters and double-byte characters) and for data that is not associated with any character set (called bit data). Note that 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) of a native encoding scheme identifies one of the coded character sets in which data can be stored at that site. A CCSID of a foreign encoding scheme identifies 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 coded character set with an EBCDIC or Unicode encoding scheme, but not in an ASCII encoding scheme.

A variable containing data in a foreign encoding scheme 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. The rules are product-specific.

For details on character conversion, see:

- "Conversion rules for assignments" on page 82
- "Conversion rules for comparison" on page 88
- "Conversion rules for operations that combine strings" on page 95
- "Considerations for using distributed relational database" on page 976.

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 a length is an attribute of strings. All values of the same string column have the same CCSID. In DB2 for LUW, support for CCSIDs is limited to DRDA®. CCSIDs are mapped into code page identifiers when receiving DRDA flows and code page identifiers are mapped into CCSIDs when sending DRDA flows.

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 each 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. Which type of conversion is used for a specific source and target CCSID is product-specific. 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 method of specifying the default CCSID(s) is product-specific.

The CCSID of the following types of string values is determined by using the default CCSID at the current server:

- String constants (including string constants that represent datetime values)
- Special registers with string values (such as USER and CURRENT SERVER)
- CAST specifications where the result is a character or graphic string
- The result of the CHAR, CLOB, DAYNAME, DBCLOB, DECRYPT_CHAR, DIGITS, GETHINT, GRAPHIC, HEX, MONTHNAME, SOUNDEX, SPACE, VARCHAR, VARCHAR_FORMAT, and VARGRAPHIC scalar functions
- Character and graphic string columns defined by CREATE TABLE and ALTER TABLE statements.

- Character and graphic string columns defined by DECLARE GLOBAL TEMPORARY TABLE statements.
- Distinct types when the source type is a character or graphic string type.

In a distributed application, the default CCSID of variables is determined by the application requester. In a nondistributed application, the default CCSID of variables is determined by the application server.

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. Each computer system has a relational database manager to manage the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a given 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. 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 6.

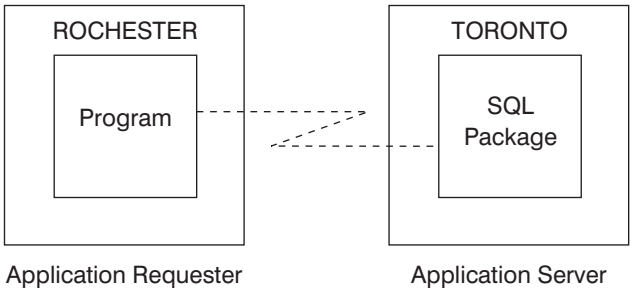


Figure 6. A distributed relational database environment

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

Application servers

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

A *connection* is an association between an application process 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 application process.

An application server can be local to, or remote from, the environment where the process is started. (An application server is present, even when not using

distributed relational databases.) This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement.
G The format and maintenance of this directory are product-specific.

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.

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 might be running via the application requester of a database manager that does not support some of those statements and clauses. Restrictions are listed in "Considerations for using distributed relational database" on page 976.

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. See "CONNECT (type 1)" on page 598.
- CONNECT (Type 2) is used for application-directed distributed unit of work. See "CONNECT (type 2)" on page 602.

See "CONNECT (type 1) and CONNECT (type 2) differences" on page 979 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 application process at computer system A can connect to an application server at computer system B and, within one or more units of work, execute any number of static or dynamic SQL statements that reference objects at B. After ending a unit of work at B, the application process 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 application process 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: [9]

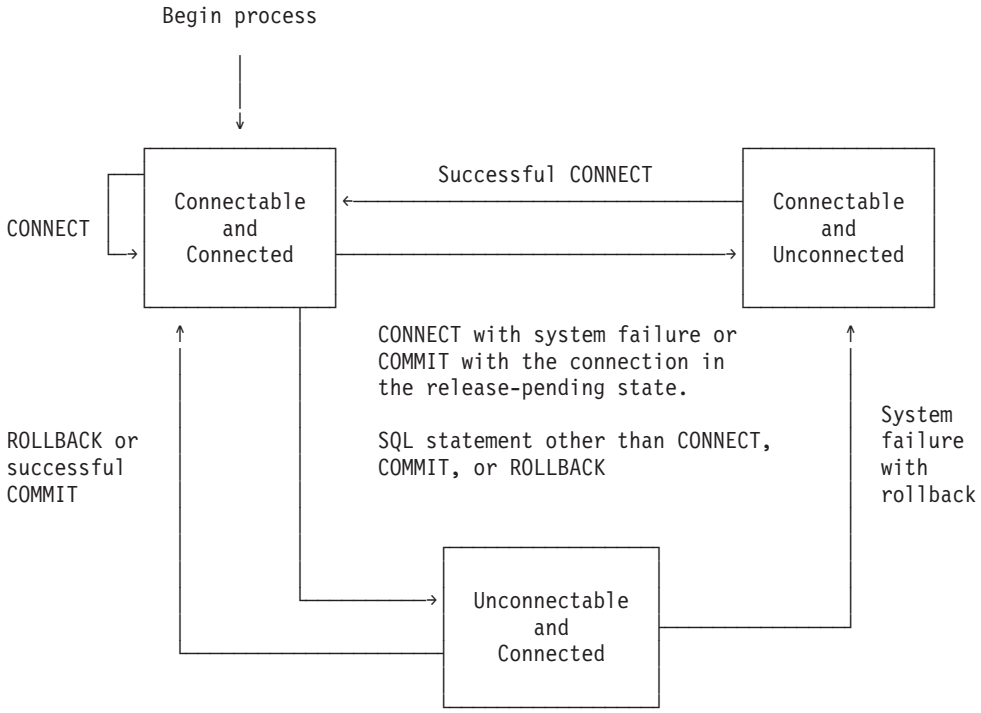


Figure 7. State transitions for an application process connection in a remote unit of work

G The initial state of an application process is *connectable* and *connected*. The
G application server to which the application process is connected is determined by a
G product-specific option that may involve an implicit CONNECT operation. An implicit CONNECT operation cannot occur if an implicit or explicit CONNECT operation has already successfully or unsuccessfully occurred. Thus, an application process cannot be implicitly connected to an application server more than once.
G The other rules for implicit CONNECT operations are product-specific.

The connectable and connected state: An application process is connected to an application server and CONNECT statements can be executed. The process 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 application process is connected to an application server, but a CONNECT statement cannot be successfully executed to change application servers. The process 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 application process is not connected to an application server. The only SQL statement that can be executed is CONNECT.

The application process enters this state when:

9. Rollback in this diagram and following discussion refers to ROLLBACK without the TO SAVEPOINT clause.

- The connection was in a connectable state, but the CONNECT statement was unsuccessful.
- The connection was in a release-pending state, and a COMMIT operation is performed.

G The other reasons for entering this state are product-specific.

G In DB2 for z/OS, an application process can also be in the *unconnectable* and
G *unconnected* state. An application process enters this state as a result of a system
G failure that has caused a rollback at the application server. An application process
G in this state must execute a rollback operation.

Consecutive CONNECT statements can be executed successfully because CONNECT does not remove the application process from the connectable state. A CONNECT to the application server to which the application process 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, RELEASE, ROLLBACK, or SET CONNECTION. 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* [10] *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 application process 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.

Application-directed distributed unit of work connection management

At any time:

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

Initial state of an application process: An application process is initially in the connected state and has exactly one connection. The initial states of a connection are current and held.

The following diagram shows the state transitions:

10. For DB2 for z/OS, the term used is *DRDA access* where the application issues explicit CONNECT statements.

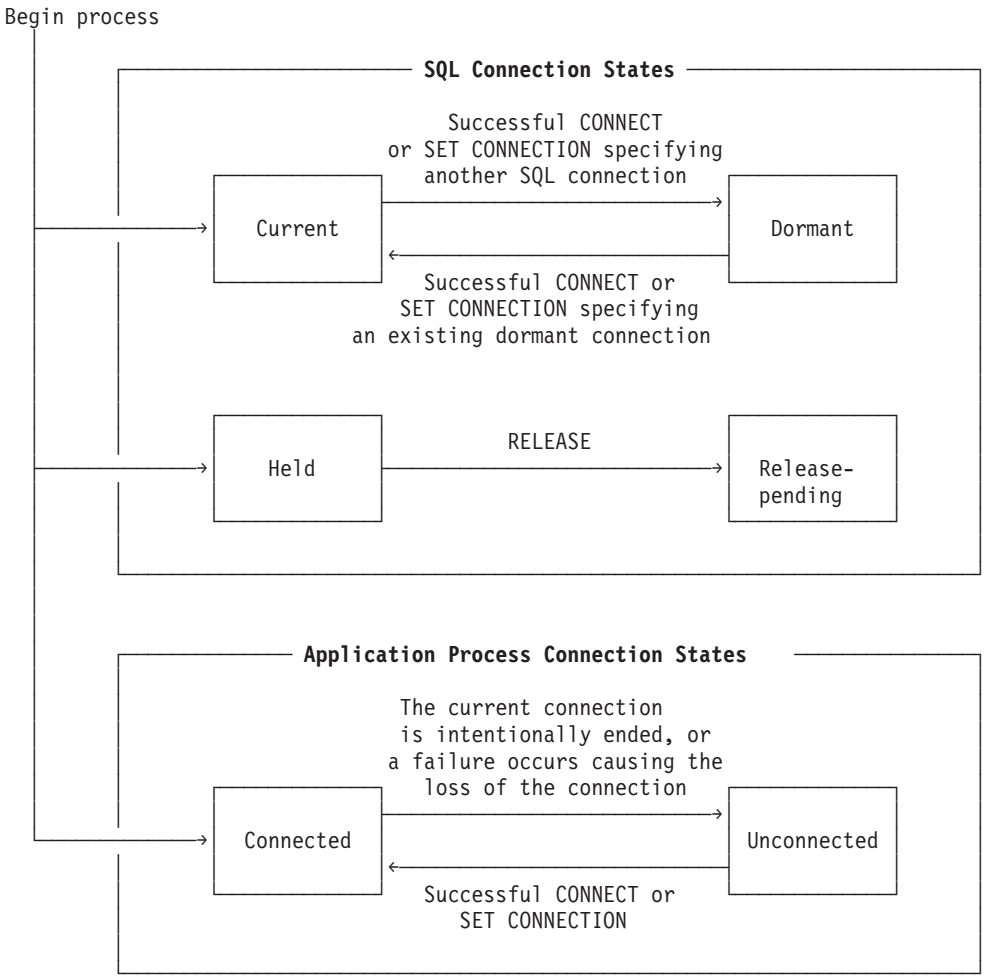


Figure 8. State transitions for an application process connection in an application-directed distributed unit of work

Connection states

If an application process successfully executes a CONNECT statement:

- The current connection is placed in the dormant and held state, and
- 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 application process, an error is returned.

A connection in the dormant state is placed in the current state using the SET CONNECTION statement. [11] 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 application process 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.

11. Some products provide a product-specific option that allows the CONNECT statement to place a connection in the dormant state.

A connection is placed in the release-pending state by the RELEASE statement. When an application process executes a commit operation, every release-pending connection of the process 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.

Application process connection states

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

- An application process cannot have more than one connection to the same application server at the same time.
- When an application process executes a SET CONNECTION statement, the specified location name must be an existing connection in the set of connections of the application process.
- When an application process executes a CONNECT statement, the specified server name must not be an existing connection in the set of connections of the application process. [12]

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

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

If an application process does not have a current connection, the application process 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, SET CONNECTION, RELEASE, COMMIT, and ROLLBACK.

An application process 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 application process successfully executes a commit operation and the connection is in the release-pending state.

When a connection is ended

When a connection is ended, all resources that were acquired by the application process 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 release-pending 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 application process is placed in the unconnected state. All connections of an application process are ended when the application process ends.

12. In DB2 for z/OS, this rule is enforced only if the SQLRULES(STD) bind option is specified.

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 of the data and the environment type of the sending system. For example, when a floating-point variable from a DB2 for i application requester is assigned to a column of a table at an DB2 for z/OS application server, DB2 for z/OS, knowing the data type and the sending system, converts the number from IEEE format to S/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 that is to be performed with that data. Strings are converted in accordance with the IBM Character Data Representation Architecture (CDRA). For more information on character conversion, refer to *Character Data Representation Architecture Reference and Registry*, SC09-2190.

Chapter 3. Language elements

This chapter 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.

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

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[13]	^	caret
!	exclamation mark	[	left bracket
{	left brace	]	right bracket
}	right brace		

13. Using the vertical bar (|) character might inhibit code portability between IBM relational database products. Use the CONCAT operator in place of the || operator.

Tokens

The basic syntactic units of the language are called *tokens*. A token consists of one or more characters, excluding the blank character 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 833.

  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 1. Control Characters

Control Character	EBCDIC Hex Value	ASCII and UTF-8 Hex Value	UCS-2 and UTF–16 Hex Value
Tab	05	09	U+0009
Form Feed	0C	0C	U+000C
Carriage Return	0D	0D	U+000D
New Line	15	85	U+0085
Line Feed (New line)	25	0A	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.

Comments: Dynamic SQL statements may include SQL comments. Static SQL statements may include host language comments or SQL comments. For more information on host language comments see the Host Language Appendices. 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. See Chapter 17, "Coding SQL statements in Java applications," on page 1099.

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 528.

bracketed comments
Bracketed comments are introduced with /* and end with */. A bracketed comment can continue past the end of the line. For additional information, see "SQL comments" on page 528.

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 EMPLOYEE where lastname = 'Smith';
```

is equivalent, after folding, to:

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

Identifiers

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

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. An ordinary identifier should not be a reserved word. See "Reserved words" on page 1168 for a list of reserved words. If a reserved word is used as an identifier in SQL, it must be specified in uppercase and must be a delimited identifier or specified in a variable.
- A *delimited identifier* is a sequence of characters enclosed within quotation marks ("). The sequence must consist of one or more characters of the SQL language. 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 quotation marks.

Examples

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

See Table 2 on page 50 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. In non-Java programs, do not use names beginning with `'DB2'`, `'SQ'`[14], `'SQL'`, `'sql'`, `'RDI'`, or `'DSN'` because precompilers generate variable names that begin with these characters. In Java, do not use names beginning with `'__sJT_'`.

See Table 2 on page 50 for the limits on the maximum size of the host identifier name imposed by each product.

14. `'SQ'` is allowed in C, COBOL, and REXX.

Naming conventions

The rules for forming a name depend on the type of the object designated by the name. Many database objects have a *schema qualified name*. A *schema qualified name* may consist of a single SQL identifier (in which case the *schema-name* is implicit) or a *schema-name* followed by a period and an SQL identifier. The following list defines these terms.

alias-name
A qualified or unqualified name that designates an alias. The unqualified form of *alias-name* is an SQL identifier. An unqualified *alias-name* in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

array-type-name
A qualified or unqualified name that designates an array type. The unqualified form of *array-type-name* is an SQL identifier. An unqualified *array-type-name* in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the *array-type-name* appears as described by the rules in "Unqualified function, procedure, specific name, type, and variables" on page 52. The qualified form is a *schema-name* followed by a period and an SQL identifier.

authorization-name
An SQL identifier that designates a user or group of users. An *authorization-name* must not be a delimited identifier that includes lowercase letters or special characters. See "Authorization IDs and authorization names" on page 55 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 view. The unqualified form of *column-name* is an SQL identifier. The qualified form is a qualifier followed by a period and an SQL identifier. The qualifier is a qualified or unqualified table or view name, or a correlation name.

constraint-name
An SQL identifier that designates a check, primary key, referential, or unique constraint on a 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. In SQLJ, *cursor name* is a variable (with no indicator variable) that identifies an instance of an iterator.

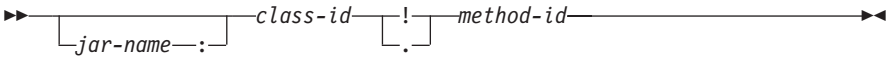
descriptor-name
A variable name that designates an SQL descriptor area (SQLDA). See "References to host variables" on page 116 for a description of a variable. Note that *descriptor-name* never includes an indicator variable. In C, the *descriptor-name* must be a pointer. For more information, see "Using pointer data types in C" on page 1077.

distinct-type-name
A qualified or unqualified name that designates a distinct type. The unqualified form of *distinct-type-name* is an SQL identifier. An unqualified

distinct-type-name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the *distinct-type-name* appears as described by the rules in "Unqualified function, procedure, specific name, type, and variables" on page 52. The qualified form is a *schema-name* followed by a period and an SQL identifier.

external-program-name
There are two distinct forms of an *external- program-name* that designate an external program.

- In C and COBOL, *external-program-name* is an SQL identifier. The SQL identifier must not be a delimited identifier or include the underscore character.
- In Java, *external-program-name* is a character string. The format of the character string is 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
A case-sensitive string that designates a JAR.

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 myPackage/ directory for the StoredProcs classes.

For details regarding the location or installation of Java classes, see the product documentation.

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 function. The unqualified form of *function-name* is an SQL identifier. An unqualified function-name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the function appears as described by the rules in "Unqualified function, procedure, specific name, type, and variables" on page 52. The qualified form is a *schema-name* followed by a period and an SQL identifier.

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 116.

index-name
A qualified or unqualified name that designates an index. The unqualified

form of an *index-name* is an SQL identifier. An unqualified *index-name* in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

mask-name
A qualified or unqualified name that designates a mask. The unqualified form of a *mask-name* is an SQL identifier. An unqualified *mask-name* in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

package-name
A qualified or unqualified name that designates a package. The unqualified form of *package-name* is an SQL identifier. A *package-name* must not be a delimited identifier that includes lowercase letters or special characters. An unqualified package-name in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

G In DB2 for z/OS, a package-name in an SQL statement must be qualified.

parameter-name
An SQL identifier that designates a parameter in an SQL procedure or SQL function.

partition-name
An unqualified identifier that designates a data partition of a partitioned
G table. In DB2 for z/OS, an integer must be specified for partition-name.

permission-name
A qualified or unqualified name that designates a permission. The unqualified form of a *permission-name* is an SQL identifier. An unqualified *permission-name* in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

procedure-name
A qualified or unqualified name that designates a procedure. The unqualified form of *procedure-name* is an SQL identifier. The implicit qualifier is a schema name, which is determined by the context in which the procedure appears as described by the rules in "Unqualified function, procedure, specific name, type, and variables" on page 52. The qualified form is a *schema-name* followed by a period and an SQL identifier.

savepoint-name
An unqualified SQL identifier that designates a savepoint.

schema-name
An SQL identifier that provides a logical grouping for SQL objects. A *schema-name* is used as the qualifier of the name of SQL objects (see "Reserved schema names" on page 1167).

G In DB2 for i, a blank is not allowed in a delimited schema name.

sequence-name
A qualified or unqualified name that designates a sequence. The unqualified form of a *sequence-name* is an SQL identifier. The unqualified form is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

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.

specific-name
A qualified or unqualified name that designates a function or procedure. The unqualified form of *specific-name* is an SQL identifier. The implicit qualifier is a schema name, which is determined by the context in which the specific name appears as described by the rules in "Unqualified function, procedure, specific name, type, and variables" on page 52. The qualified form is a *schema-name* followed by a period and an SQL identifier.

SQL-condition-name
An SQL identifier that designates a condition in an SQL procedure.

SQL-label
An unqualified name that designates a label in an SQL procedure.

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

SQL-variable-name
A qualified or unqualified name that designates a variable in the SQL routine body of an SQL procedure. 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.

table-identifier
An unqualified SQL identifier that designates a table.

table-name
A qualified or unqualified name that designates a table. The unqualified form of *table-name* is an SQL identifier. An unqualified *table-name* in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

trigger-name
A qualified or unqualified name that designates a trigger. The unqualified form of *trigger-name* is an SQL identifier. An unqualified *trigger-name* in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

view-name
A qualified or unqualified name that designates a view. The unqualified form of *view-name* is an SQL identifier. An unqualified *view-name* in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

Table 2. Identifier Length Limits (in bytes)

Identifier Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Longest authorization name	8	10^{138}	128	8
Longest constraint name	128	128	128	128
Longest correlation name	128	128	128	128

Table 2. Identifier Length Limits (in bytes) (continued)

Identifier Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Longest cursor name	128[139]	128	128	128
Longest external program name (string form)	1305	279	254	254
Longest external program name (unqualified form)	8	10	128	8
Longest host identifier[140]	128	128	255	128
Longest package version-ID	64	64	64	64
G Longest partition name	n/a	10	128	10
Longest savepoint name	128	128	128	128
Longest schema name	128	128	128[141]	128
Longest server name	16	18	8	18
Longest SQL condition name	128	128	128	128
Longest SQL label	128	128	128	128
Longest statement name	128	128	128	128
Longest unqualified alias name	128	128	128	128
Longest unqualified column name	30	128	128	30
Longest unqualified function name	128	128	128	128
Longest unqualified index name	128	128	128	128
Longest unqualified mask name	128	128	128	128
Longest unqualified package name	8	10	128	8
Longest unqualified permission name	128	128	128	128
Longest unqualified procedure name	128	128	128	128
Longest unqualified sequence name	128	128	128	128
Longest unqualified specific name	128	128	128	128
Longest unqualified SQL parameter name	128	128	128	128
Longest unqualified SQL variable name	128	128	128	128
Longest unqualified table and view name	128	128	128	128
Longest unqualified trigger name	128	128	128	128
Longest unqualified type name	128	128	128	128

SQL path

The *SQL path* is an ordered list of schema names. The database manager uses the SQL path to resolve the schema name for unqualified data 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, GRANT or REVOKE statement. For further details, see "Qualification of unqualified object names."

For example in DB2 for i, if the SQL path is SMITH, XGRAPHIC, QSYS, QSYS2 and an unqualified distinct type name MYTYPE was specified, the database manager looks for MYTYPE first in schema SMITH, then XGRAPHIC, and then QSYS and QSYS2.

The SQL path used depends on the SQL statement:

- For static SQL statements (except for a CALL *variable* statement), the SQL path used is the SQL path specified when the containing package, procedure, function, trigger, or view was created. The way the SQL path is specified is product-specific.
- For dynamic SQL statements (and for a CALL *variable* statement), the SQL path is the value of the CURRENT PATH special register. CURRENT PATH can be set by the SET PATH statement. For more information, see "SET PATH" on page 883.

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 on the SQL path for dynamic SQL, see "CURRENT PATH" on page 104.

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, index, mask, package, permission, sequence, table, trigger, and view names

Unqualified alias, index, mask, package, permission, sequence, table, trigger, and view names are implicitly qualified by the *default schema*.

- For static SQL statements, the default schema is the default schema specified when the containing mask, function, package, permission, procedure, or trigger was created. Each product provides a product-specific means of explicitly specifying a default schema.
- For dynamic SQL statements, the default schema is the default schema specified for the application process. The default schema can be specified for the application process by using the SET SCHEMA statement (See "SET SCHEMA" on page 886).

If the default schema is not explicitly specified, the default schema is the authorization ID of the statement.

Unqualified function, procedure, specific name, type, and variables

The qualification of function, procedure, specific name, type (both built-in types and user-defined types), and variables depends on the SQL statement in which the unqualified name appears:

- If an unqualified name is the main object of an ALTER, CREATE, COMMENT, DROP, GRANT, or REVOKE statement, the name is implicitly qualified using the same rules as for qualifying unqualified table names (See "Unqualified alias, index, mask, package, permission, sequence, table, trigger, and view names" on page 52).
- Otherwise, the implicit schema name is determined as follows:
  - For user-defined type names, the database manager searches the SQL path and selects the first schema in the SQL path such that the data type exists in the schema.
  - For procedure names, the database manager searches the SQL path and selects the first schema in the SQL path such that the schema contains an authorized procedure with the same name and the same number of parameters.
  - For function names, the database manager uses the SQL path in conjunction with function resolution, as described under "Function resolution" on page 124.
  - For specific names specified for sourced functions, see "CREATE FUNCTION (sourced)" on page 632.

Aliases

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

An alias can be used wherever a table or view name can be used. However, 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 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 an alias is created. An alias cannot refer to another alias.

The option of referring to a table or view 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, 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 or view 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 statements 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.
- G The effect on any views, materialized query tables, routines, or triggers that G reference the alias is product-specific.

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.

Authorization IDs 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 applies to a static SQL statement is the authorization ID that is used during program preparation. The authorization ID that applies to a dynamic SQL statement is the authorization ID that was obtained by the database manager when a connection was established between the database manager and the process.[15] This is called the *run-time authorization ID*.

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.

Example

Assume SMITH is the user ID and the authorization ID that the database manager obtained when the connection was established with the application process. The following statement is executed 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 and SMITH is the default schema.

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

15. In DB2 for z/OS and DB2 for LUW the DYNAMICRULES bind option setting can impact the authorization ID that applies to a dynamic SQL statement. For details, refer to product specific documentation.

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 70.

Figure 9 on page 57 illustrates the various data types supported by the DB2 relational database products:

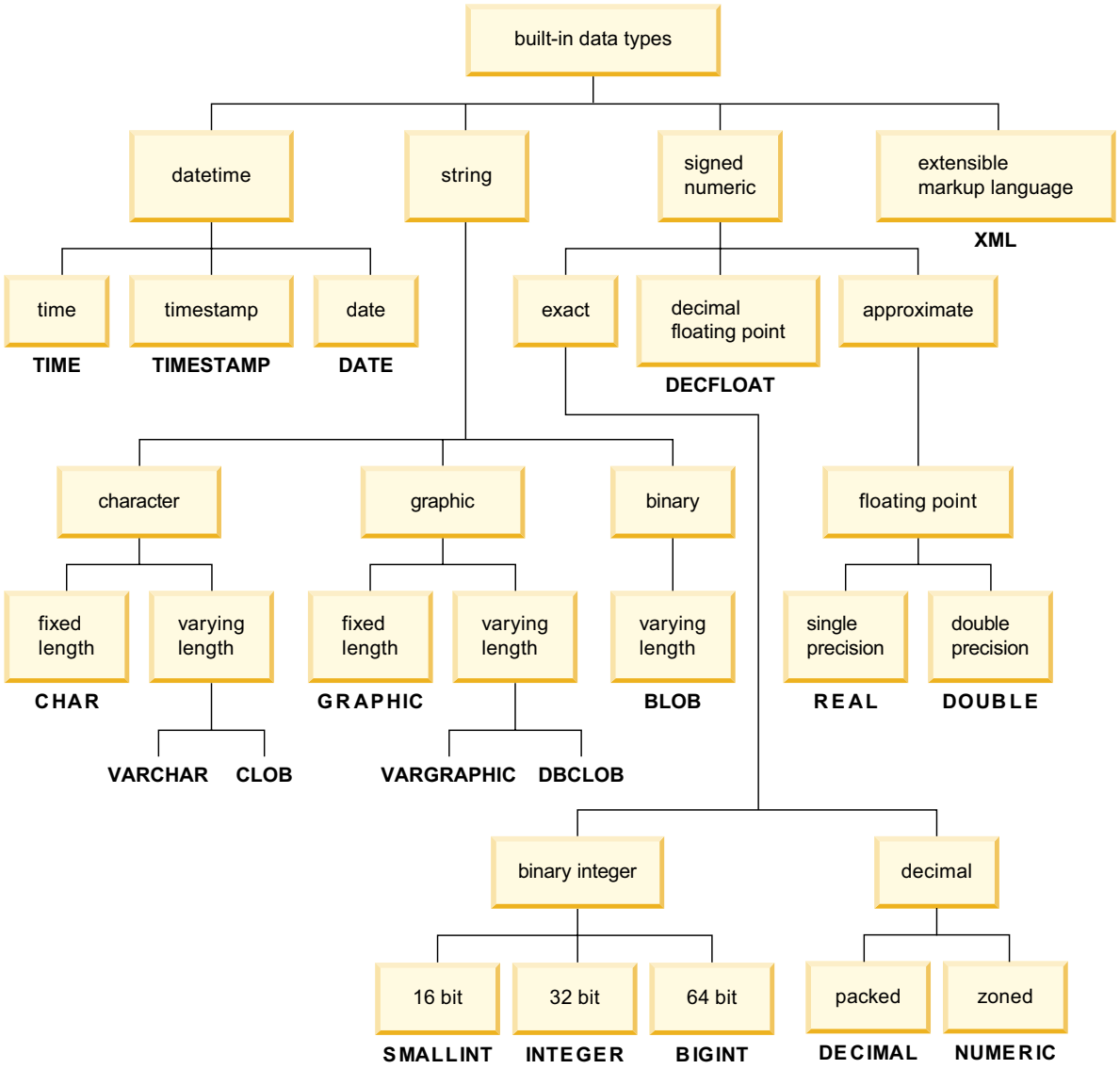


Figure 9. Data Types Supported by the DB2 Relational Database Products

G In DB2 for z/OS, and DB2 for LUW, zoned decimal is not supported as a native
G data type and NUMERIC is treated as a synonym for DECIMAL. Zoned decimal
G numbers received through DRDA protocols are converted to packed decimal.

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 GROUP BY, SELECT DISTINCT, and set 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.

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. 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 value of 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 for 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 binary or decimal digits excluding the sign. The scale is the total number of binary or decimal 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 and a scale of zero. The range of small integers is -32 768 to +32 767.

Large integer

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

Big integer

A *big integer* is a binary number composed of 8 bytes with a precision of 19 digits and a scale of zero. 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 31 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 $-10^{31} + 1$ to $10^{31} - 1$.

Floating-point

A *single-precision floating-point* number is a 32-bit approximate representation of a real number. The number can be zero or can range from $-3.4x10^{38}$ to $-1.17x10^{-37}$, or from $+1.17x10^{-37}$ to $+3.4x10^{38}$.

A *double-precision floating-point* number is a 64-bit approximate representation of a real number. The number can be zero or can range from $-7.2x10^{75}$ to $-5.4x10^{-79}$, or from $+5.4x10^{-79}$ to $+7.2x10^{75}$.

See Table 60 on page 964 for more information.

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 98 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.[16]

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 60 on page 964 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++, COBOL, and Java. Floating-point variables can be used in all host languages. Decimal variables can be declared in all host languages except 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. The mechanism to specify the default decimal separator character is product-specific.

16. In DB2 for i the warning is only returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

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. [16]

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.[16] 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.

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.

G DB2 for LUW does not return subnormal warnings.

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 and 254 inclusive. See Table 60 on page 964 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 and 32 672 inclusive. For a CLOB string, the length attribute must be between 1 and 2 147 483 647 inclusive. See Table 60 on page 964 for more information.

Character-string variables

- Fixed-length character-string variables can be used in all host languages except REXX and Java. (In C, fixed-length character string variables are limited to a length of 1.)
- Varying-length character-string variables can be used in all host languages except CLOBs cannot be used in REXX.

For information on how to code in a host language, refer to the host language appendices.

Character encoding schemes

Each character string is further defined as one of:

Bit data
Data that is not associated with a coded character set and is therefore never converted. The CCSID for bit data is X'FFFF' (65535). In DB2 for LUW, the CCSID for bit data is X'0000' (zero).

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

In DB2 for LUW, support for CCSIDs is limited to DRDA. CCSIDs are mapped into code page identifiers when receiving DRDA flows and code page identifiers are mapped into CCSIDs when sending DRDA flows.

The method of representing DBCS characters within a mixed string differs between ASCII and EBCDIC.

- ASCII reserves a set of code points for SBCS characters and another set as the first half of DBCS characters. Upon encountering the first half of a DBCS character, the system knows that it is to read the next byte in order to obtain the complete character.
- EBCDIC makes use of two special code points:
  - A shift-out character (X'0E') to introduce a string of DBCS characters.
  - A shift-in character (X'0F') to end a string of DBCS characters.

The default encoding scheme is specific to the relational database. Because of the shift characters, EBCDIC mixed data requires more storage than ASCII mixed data.

Examples

For the same mixed data character string, Table 3 shows character and hexadecimal representations of the character string in different encoding schemes. In EBCDIC, the shift-out and shift-in are needed to delineate the double-btye characters.

Table 3. Example of a character string in different encoding schemes

Data type and encoding schema	Character representation	Hexadecimal representation (with spaces separating each character)
9 bytes in ASCII	元gen 気ki	8CB3 67 65 6E 8B43 6B 69
13 bytes in EBCDIC	S0元SI gen S0気SI ki	0E 4695 0F 87 85 95 0E 45B9 0F 92 89
11 bytes in Unicode UTF-8	元gen 気ki	E58583 67 65 6E E6B097 6B 69

To minimize the effects of these differences, use varying-length strings with an appropriate declared length in applications that require mixed data and operate on both ASCII and EBCDIC systems.

Graphic strings

A *graphic string* is a sequence of double-byte characters. The length of the string is the number of double-byte characters in the sequence. 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 and 127 inclusive. See Table 60 on page 964 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 and 16 336 inclusive. For a DBCLOB string, the length attribute must be between 1 and 1 073 741 823 inclusive. See Table 60 on page 964 for more information.

Graphic-string variables

- Fixed-length graphic-string variables can be declared in all host languages except REXX and Java. (In C, fixed-length graphic-string variables are limited to a length of 1.)

- Varying-length graphic-string variables can be declared in all host languages except DBCLOBs cannot be used in REXX.

For information on how to code in a host language, refer to the host language appendices.

Graphic encoding schemes

Each graphic string is further defined as one of:

DBCS data
Data in which every character is represented by a character from the double-byte character set (DBCS). 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 four bytes. Each Unicode graphic string is encoded using either UCS-2 or UTF-16. Each Unicode string has an associated CCSID.

Binary strings

A *binary string* is a sequence of bytes. The length of a binary string (BLOB string) is the number of bytes in the sequence. 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 BLOB string, the length attribute must be between 1 and 2 147 483 647 bytes inclusive. See Table 60 on page 964 for more information.

A variable with a BLOB string type can be defined in all host languages except REXX.

Although BLOB strings and FOR BIT DATA character strings might be used for similar purposes, the two data types are not compatible. The BLOB function can be used to to change a FOR BIT DATA character string into a BLOB string.

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). [17]

17. 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.

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 LOB locator value as input. The resulting output of the LOB 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, 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.

Limitations on use of strings

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 4. Contexts for limitations on use of varying-length strings

Context of usage	LOB (CLOB, DBCLOB, or BLOB)
A GROUP BY clause	Not allowed
An ORDER BY clause	Not allowed

Table 4. Contexts for limitations on use of varying-length strings (continued)

Context of usage	LOB (CLOB, DBCLOB, or BLOB)
A CREATE INDEX statement	Not allowed
A SELECT DISTINCT statement	Not allowed
A subselect of a UNION without the ALL keyword	Not allowed
Predicates	Cannot be used in any predicate except EXISTS, LIKE, and NULL. This restriction includes a *simple-when-clause* in a CASE expression. *expression* WHEN *expression* in a *simple-when-clause* is equivalent to a predicate with *expression=expression*.
The definition of primary, unique, and foreign keys	Not allowed
Check constraints	Cannot be specified for a LOB column

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 66.

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. [18] The range of the year part is 0001 to 9999. 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 length of a DATE column as described in the SQLDA is 10 bytes, which is the appropriate length for a character-string representation of the value.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB.

Time

A *time* is a three-part value (hour, minute, and second) designating a time of day using a 24-hour clock. 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 length of a TIME column as described in the SQLDA is 8 bytes, which is the appropriate length for a character-string representation of the value.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB.

Timestamp

A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and optional fractional second) that represents a date and time.

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

The time portion of a timestamp value can include a specification of fractional seconds. The number of digits in the fractional seconds is specified using an attribute in the range from 0 to 12 with a default of 6.

The length of a TIMESTAMP column as described in the SQLDA is is between 19 and 32 bytes, which is the appropriate length for the character-string representation of the value.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB.

Datetime variables

Character string variables are normally used to contain date, time, and timestamp values. However, 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 strings or Unicode graphic strings. These representations directly concern the user of SQL since for many host languages there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Thus, to be retrieved, a datetime value must 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 mechanism to specify the default date format and default time format is product-specific.

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 is not the same as the default CCSID for SBCS data, the string 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 string that starts with a digit and has a length of at least 8 characters. Trailing blanks can be included. Leading zeros can be omitted from the month and day portions. Valid string formats for dates are listed in Table 5. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use.

Table 5. Formats for String Representations of Dates

Format Name	Abbreviation	Date Format	Example
International Standards Organization	ISO	yyyy-mm-dd	1987-10-12
IBM USA standard	USA	mm/dd/yyyy	10/12/1987
IBM European standard	EUR	dd.mm.yyyy	12.10.1987
Japanese industrial standard Christian era	JIS	yyyy-mm-dd	1987-10-12

Time strings: A string representation of a time is a character 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 seconds are omitted, an implicit specification of 0 seconds is assumed. Thus, 13:30 is equivalent to 13:30:00. Although all products accept times of 24:00:00, the handling of such times during arithmetic operations is product-specific.

Valid string formats for times are listed in Table 6. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use.

Table 6. Formats for String Representations of Times

Format Name	Abbreviation	Time Format	Example
International Standards Organization	ISO	hh.mm.ss [19]	13.30.05
IBM USA standard	USA	hh:mm AM or PM	1:30 PM
IBM European standard	EUR	hh.mm.ss	13.30.05
Japanese industrial standard Christian era	JIS	hh:mm:ss	13:30:05

The following additional rules apply to the USA time 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:

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

Timestamp strings: A string representation of a timestamp is a character string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form *yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn*. Trailing blanks can be included. Leading zeros can be omitted from the month, day, and hour part of the timestamp. Trailing zeros can be truncated or omitted entirely from fractional seconds. If any trailing digit of the fractional seconds portion is omitted, an implicit specification of 0 is assumed. Thus, *1990-3-2-8.30.00.10* is equivalent to *1990-03-02-08.30.00.100000000000*. Although all products accept timestamps whose time part is 24.00.00.000000000000, the handling of such timestamps during arithmetic operations is product-specific.

19. This is an earlier version of the International Standards Organization format. The JIS format is equivalent to the current International Standards Organization format.

SQL statements also support the ODBC or JDBC string representation of a timestamp as an input value only. The ODBC and JDBC string representation of a timestamp has the form yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn.

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. Unlike the LOB data type which has a LOB locator type, there is no XML locator type.

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, HEX, LENGTH, CONTAINS, and SCORE scalar functions
- XML scalar functions
- A *select-clause* without DISTINCT
- INSERT VALUES clause, UPDATE SET clause, and MERGE
- SET variable and VALUES INTO
- Procedure parameters
- User-defined function arguments and result
- 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, IN, or LIKE predicate
- An aggregate function with the DISTINCT keyword
- A primary, unique, or foreign key
- A check constraint
- An index column
- Trigger transition variables

- CREATE TYPE statements

No host languages have a built-in data type for the XML data type.

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 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 734.

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 types allows functions to be created specifically for AUDIO and assures that these functions cannot be applied to other 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, or the object of a DROP, COMMENT, GRANT, or REVOKE statement, the normal process of qualification by the default schema is used to determine the schema name.
- In any other context, the SQL path is used to determine the schema name. The schemas in the SQL path are searched, in sequence, and the first schema in the SQL path is selected such that the distinct type exists in the schema and the user has authorization to use the type. For a description of the SQL path, see "SQL path" on page 52.

A distinct type does not automatically acquire the functions and operators of its source type because they might not be meaningful. (For example, it might make sense for a length function for an AUDIO type to return the length 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 on 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.

The comparison operators are automatically generated for distinct types, except those that are sourced on a CLOB, DBCLOB, or BLOB. In addition, functions are generated for every 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, a precedence order 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 to VARCHAR.

The database manager considers the promotion of data types when:

- performing function resolution (see "Function resolution" on page 124)
- casting distinct types (see "Casting between data types" on page 74)
- assigning built-in data types to distinct types(see "Distinct type assignments" on page 84).

For each data type, Table 7 shows the precedence list (in order) that the database manager uses to determine the data types to which a given 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 7. 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
BLOB	BLOB
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
XML	XML
udt	same udt

Table 7. Data Type Precedence Table (continued)

Data Type	Data Type Precedence List (in best-to-worst order)

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. Character bit data and graphic strings are not compatible.

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 defined in "Promotion of data types" on page 72) is one example 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, CAST specification (see "CAST specification" on page 149), or XMLCAST specification (see "XMLCAST specification" on page 162) 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 84). In addition, when a sourced user-defined function is created, the data types of the parameters of the source function must be castable to the data types of the function that is being created (see "CREATE FUNCTION (sourced)" on page 632).

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 81).

If truncation occurs when casting to a binary string, an error is returned.

For casts that involve a distinct type as either the data type to be cast to or from, Table 8 shows the supported casts. For casts between built-in data types, Table 9 on page 75 shows the supported casts.

Table 8. 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 72)
INTEGER	Distinct type *DT* if *DT*'s source data 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

Character and graphic strings are only compatible for Unicode data. Character bit data and graphic strings are not compatible.

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 more information, see "Function resolution" on page 124, and "CAST specification" on page 149. 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 9. 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	BLOB	DATE	TIME	TIMESTAMP	XML
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
NUMERIC	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
DECFLOAT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y[1]	Y[1]	-	Y	Y	Y	Y	-
VARCHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y[1]	Y[1]	-	Y	Y	Y	Y	-
CLOB	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	Y[1]	Y	-	-	-	-
GRAPHIC	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	-	-	Y[1]	Y[1]	Y[1]	Y[1]	Y	Y	Y	Y	Y[1]	Y[1]	Y[1]	-
VARGRAPHIC	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	-	-	Y[1]	Y[1]	Y[1]	Y[1]	Y	Y	Y	Y	Y[1]	Y[1]	Y[1]	-
DBCLOB	-	-	-	-	-	-	-	-	Y[1]	Y[1]	Y[1]	Y	Y	Y	Y	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-
DATE	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	Y	-	-	-
TIME	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-
TIMESTAMP	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	Y	Y	Y	-
XML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y

Notes

[1] The cast is only allowed if the encoding scheme of the graphic-string data type is Unicode and the character string data type is not bit data.

The following table indicates where to find the rules that apply for each cast:

Table 10. Rules for Casting to a Data Type

Target Data Type	Rules
SMALLINT	See “SMALLINT” on page 369.
INTEGER	See “INTEGER or INT” on page 289.
BIGINT	See “BIGINT” on page 221.
DECIMAL	See “DECIMAL or DEC” on page 257.
NUMERIC	See “DECIMAL or DEC” on page 257.
REAL	See “REAL” on page 344.
DOUBLE	See “DOUBLE_PRECISION or DOUBLE” on page 265.

Table 10. Rules for Casting to a Data Type (continued)

Target Data Type	Rules
DECFLOAT	See “DECFLOAT” on page 255.
CHAR	See “CHAR” on page 227.
VARCHAR	See “VARCHAR” on page 404.
CLOB	See “CLOB” on page 234.
GRAPHIC	See the rules for string assignment to a variable in “Assignments and comparisons” on page 77.
VARGRAPHIC	See the rules for string assignment to a variable in “Assignments and comparisons” on page 77.
DBCLOB	See “DBCLOB” on page 253.
BLOB	See “BLOB” on page 224.
DATE	See “DATE” on page 245.
TIME	See “TIME” on page 382.
TIMESTAMP	If the source data type is a character string, see “TIMESTAMP” on page 383, using the precision of the target data type as the second argument.
XML	“XMLCAST specification” on page 162

Assignments and comparisons

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of statements such as CALL, INSERT, UPDATE, FETCH, SELECT INTO, and VALUES INTO. 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 and functions such as COALESCE and CONCAT. The compatibility matrix is as follows:

Operands	Binary Integer	Decimal Number	Floating Point	Decimal Floating Point	Character String	Graphic String	Binary String	Date	Time	Timestamp	XML	Distinct Type
Binary Integer	Y	Y	Y	Y	5	6	-	-	-	-	-	4
Decimal Number	Y	Y	Y	Y	5	6	-	-	-	-	-	4
Floating Point	Y	Y	Y	Y	5	6	-	-	-	-	-	4
Decimal Floating Point	Y	Y	Y	Y	5	6	-	-	-	-	-	4
Character String	5	5	5	5	Y	1	2	3	3	3	-	4
Graphic String	6	6	6	6	1	Y	-	-	-	-	-	4
Binary String	-	-	-	-	2	-	Y	-	-	-	-	4
Date	-	-	-	-	3	-	-	Y	-	-	-	4
Time	-	-	-	-	3	-	-	-	Y	-	-	4
Timestamp	-	-	-	-	3	-	-	-	-	Y	-	4
XML[7]	-	-	-	-	-	-	-	-	-	-	Y	-
Distinct Type	4	4	4	4	4	4	4	4	4	4	-	4

Notes:

1. Bit data and graphic strings are not compatible. For DB2 for LUW, character strings and graphic strings are compatible only in a Unicode database.
2. No character strings, even those that are defined with the FOR BIT DATA attribute, are compatible with binary strings.
3. The compatibility of datetime values and character strings is limited to:
   - Datetime values can be assigned to character-string columns and to character-string variables as explained in "Datetime assignments" on page 83.
   - A valid string representation of a date can be assigned to a date column or compared with a date.
   - A valid string representation of a time can be assigned to a time column or compared with a time.
   - A valid string representation of a timestamp can be assigned to a timestamp column or compared with a timestamp.
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 84.
5. LOBs and bit data are not supported.
6. LOBs are not supported. Allowed only if the encoding scheme of the graphic-string data type is Unicode.
7. 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 116 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 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.

  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

20. In DB2 for i the warning is only returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

the rounding mode, results in zero or the smallest positive number or the largest negative number that can be represented along with a warning. [20]

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 31 and scale of 31 - (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.

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 103.

Assignments to COBOL integers

Assignment to COBOL integer variables uses the full size of the integer. Thus, the value placed in the COBOL data item field may be out of the range of values.

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.
```

Assignments from 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 149.

String assignments

There are two types of string assignments:

- *Storage assignment* is when a value is assigned to a column, a parameter of a function or procedure, or a transition variable.
- *Retrieval assignment* is when a value is assigned to a variable (but not a parameter or transition variable).

Binary string assignments

Storage assignment: The basic rule is that the length of a string assigned to a column, parameter of a function or procedure, or transition variable must not be greater than the length attribute of the column, parameter, or transition variable. If the string is longer than the length attribute of that column, parameter, or transition variable, an error is returned.

Retrieval assignment: The length of a string assigned to a variable (but not a parameter or transition 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, a warning is returned (SQLSTATE 01004) and the value 'W' is assigned to the SQLWARN1 field of the SQLCA. For a description of the SQLCA, see Chapter 11, "SQLCA (SQL communication area)," on page 981.

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 both the source and the target are strings. When a datetime data type is involved, see "Datetime assignments" on page 83. 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 84.

Assignments from 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 149.

Storage assignment: The basic rule is that the length of a string assigned to a column, parameter of a function or procedure, or transition variable must not be greater than the length attribute of the column, parameter, or transition variable. Trailing blanks are included in the length of the string. When the length of the string is greater than the length attribute of the column, parameter, or transition variable, one of the following occurs:

- the string is assigned and trailing blanks are truncated to fit the length attribute of the target column or parameter. For DB2 for z/OS, for OUT parameters, if the length of the string including trailing blanks is greater than the length attribute of the parameter, then an error is returned.
- the string is not assigned and an error is returned because truncation to fit the length attribute of the column or parameter would remove non-blank characters.

When a string is assigned to a fixed-length 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 blanks. The pad character is always a blank, even for columns defined with the FOR BIT DATA attribute.

Retrieval assignment: The length of a string assigned to a variable (but not a parameter or transition variable) can be greater than the length attribute of the variable. When a string is assigned to a variable and the length of the string is greater than the length attribute of the variable, the string is truncated on the right by the necessary number of characters. When this occurs, a warning is returned and the value 'W' is assigned to the SQLWARN1 field of the SQLCA. Furthermore, if an indicator variable is provided and the source of the value is not a LOB, the indicator variable is set to the original length of the string. The truncation result of an improperly formed mixed string is unpredictable.

When a character 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 to the right with the necessary number of blanks. The pad character is always a blank, even for strings defined with the FOR BIT DATA attribute.

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: Assignment of a character string to a variable can result in truncation of the mixed data string. Truncation removes complete characters from the right side of the mixed data string. Removal of a character that is longer than a single byte may cause the length of the result string to be less than the length attribute of the variable. If padding is then required, the single-byte blank character is used.

Assignments to C NUL-terminated strings: When a fixed-length or varying-length string of length *n* is assigned to a C NUL-terminated string variable with a length greater than *n+1*, 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 a NUL is placed in the next character position. [21]

G In DB2 for z/OS, if a varying-length string is assigned to a C NUL-terminated
G string, the value of a varying-length string column is assigned to the first *n*
G character positions of the variable, and a NUL is placed in the next character
G 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 conditions are true:

- The CCSIDs are different.
- Neither CCSID identifies bit data.
- 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 35.

An error is returned if:

- Conversion between the pair of CCSIDs is not defined. For more information, see "Coded character sets and CCSIDs" on page 35.
- A character of the string cannot be converted, and the operation is an assignment to a column or assignment to a host variable without an indicator variable. For example, a DBCS character cannot be converted to a variable with an 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 nullable 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.

G In DB2 for LUW, if a character of the string cannot be converted, an error is
G returned regardless of whether an indicator variable is provided.

21. In DB2 for i and DB2 for LUW, a program preparation option must be used for the padding and NUL placement to occur as described. For DB2 for i use the program preparation option *CNULRQD. For DB2 for LUW, use the program preparation option LANGLEVEL SQL92E. For DB2 for z/OS, use the program preparation option PADNTSTR.

G In a DB2 for LUW application server in DRDA, input variables are converted to
G the code page of the application server, even if they are assigned, compared, or
G combined with a column that is defined as FOR BIT DATA. If the SQLDA has been
G modified to identify the input variable as FOR BIT DATA, conversion is not
G performed.

Datetime assignments

| A value assigned to a DATE column or a DATE variable must be a date or a valid string representation of a date. A date can be assigned only to a DATE column, a character-string column, a DATE variable, or a character-string variable.

| A value assigned to a TIME column or a TIME variable must be a time or a valid string representation of a time. A time can be assigned only to a TIME column, a character-string column, a TIME variable, or a character-string variable.

| A value assigned to a TIMESTAMP column or a TIMESTAMP variable must be a timestamp or a valid string representation of a timestamp. A timestamp can be assigned only to a TIMESTAMP column, a character-string column, a TIMESTAMP variable, or a character-string variable.

The assignment must not be to a CLOB or DBCLOB variable or column.

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 fixed length character string 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 not a host variable and has a character data type, truncation is not allowed (for example, the target could be a column, SQL variable or SQL parameter). The length attribute of the target must be at least 10 for a date, 8 for
| a time, and 19 for a timestamp.
- When the target is a string host variable, the following rules apply:

  DATE

  The length of the variable must not be less than 10.

  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, or JIS 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.
    - SQLWARN1 is set to 'W'.
    - If an indicator variable is provided, the seconds part of the time is assigned to the indicator variable.
    - 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 31, the timestamp is truncated like a string, causing the omission of

one or more digits of fractional seconds. 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.
- **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.
- **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.

  If the source data is graphic data, the encoding scheme must be Unicode.

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 variable, 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.

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 host variable HV_AGE because the source data type of the distinct type (SMALLINT) is assignable to the host variable (INTEGER). If distinct type AGE had been sourced on a character data type such as CHAR(5), the preceding assignment would be invalid because a character type cannot be assigned to an integer type.

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 74 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 DECIMAL(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 11 shows whether the assignments are valid. The database manager uses assignment rules in this INSERT statement to determine if X can be assigned to Y.

```
INSERT INTO TABLE1(Y)
  SELECT X FROM TABLE2;
```

Table 11. Assessment of various assignments for the example INSERT statement

TABLE2.X	TABLE1.Y	Valid	Reason
AGECOL	AGECOL	Yes	Source and target are the same distinct type.
SMINTCOL	AGECOL	Yes	SMALLINT can be cast to AGE.
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	AGECOL can be cast to its source type of 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 source value for 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 that is the target of the assignment.
- If the source value for 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 that is the target of the assignment.

Assignments to LOB locators

When a LOB locator is used, it can only refer to LOB data. If a LOB locator is used for the first fetch of a cursor, LOB locators must be used for all subsequent fetches.

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 numbers with different scales are compared, the comparison is made as if one of the numbers 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 other number that has been converted to double-precision floating point. However, if a single-precision floating-point number is compared to a floating-point constant, 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.

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 cast to numeric. The string must contain a valid string representation of a number. The data type the string is cast to is product-specific.

String comparisons

Binary string comparisons

In general, comparisons that involve binary strings (BLOBs) are not supported, with the exception of the LIKE, EXISTS, and NULL predicates.

Character and graphic string comparisons

Two character or graphic strings are compared by comparing the corresponding bytes of each character or graphic string. If the strings do not have the same length, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks so that it has the same length as the other string. The pad character is always a blank, even for bit data.

The relationship between two unequal strings is determined by a comparison of the first pair of unequal bytes (or the weighted values) from the left end of the string. This comparison is made according to the collating sequence in effect when the statement is executed.

Note that the encoding scheme used for the data affects the default collating sequence, which impacts the resulting order. [22]

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 for LUW default collating sequence for United States English by showing a list that is sorted according to each one.

22. Product-specific options are available to specify a collating sequence that is different from the default collating sequence.

Table 12. Collating sequence differences

ASCII and Unicode	EBCDIC	DB2 for 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 479 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 conditions are true:

- The CCSIDs of the two strings are different.
- Neither CCSID is X'FFFF'.
- 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 35.

If a Unicode string and a non-Unicode string are compared, any necessary conversion applies to the non-Unicode string. If an SBCS string and a MIXED string are compared, any necessary conversion applies to the SBCS string. 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 13. Selecting the operand for character conversion

	Second Operand				
First Operand	**Column Value**	**Derived Value**[23]	**Constant**	**Special Register**	**Variable**
Column Value	second	second	second	second	second
Derived Value[23]	first	second	second	second	second
Constant	first	first	second	second	second
Special Register	first	first	second	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 preceding 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 35. 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 a value of that data type. All comparisons are chronological, which means the further a point in time is from January 1, 0001, the *greater* the value of that point in time.

Comparisons that involve 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 evaluated according to the following rules:

- When comparing timestamp values with different precisions, the higher precision is used for the comparison and any missing digits for fractional seconds are assumed to be zero.
- When comparing a timestamp value with a string representation of a timestamp, the string representation is first converted to TIMESTAMP(12).
- Timestamp comparisons 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.

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:

23. In DB2 for z/OS, derived values are converted before constants and special registers.

```
SELECT * FROM CAMP_DB2_ROSTER
  WHERE AGE > HIGH_SCHOOL_LEVEL
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER
  WHERE AGE > ATTENDEE_NUMBER
```

INCORRECT

However, AGE can be compared to ATTENDEE_NUMBER by using a cast function or CAST specification to convert 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
```

Array type comparisons

Comparisons of array type values are not supported. Elements of arrays can be compared based on the comparison rules for the data type of the elements.

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 function COALESCE or 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 131. For the result data type of expressions that involve the CONCAT operator, see "With the concatenation operator" on page 134.

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 only with other numeric types.

If one operand is...	And the other operand is...	The data type of the result is...
SMALLINT	SMALLINT	SMALLINT
INTEGER	SMALLINT or INTEGER	INTEGER
BIGINT	SMALLINT, INTEGER, or BIGINT	BIGINT
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where p = min(31,x+max(w-x,5))
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where p = min(31,x+max(w-x,11))
DECIMAL(w,x)	BIGINT	DECIMAL(p,x) where p = min(31,x+max(w-x,19))

If one operand is...	And the other operand is...	The data type of the result is...
DECIMAL(w,x)	DECIMAL(y,z) or NUMERIC(y,z)	DECIMAL(p,s) where p = min(31,max(x,z)+max(w-x,y-z)) and s = max(x,z)
NUMERIC(w,x)	SMALLINT	NUMERIC(p,x) where p = min(31,x+max(w-x,5))
NUMERIC(w,x)	INTEGER	NUMERIC(p,x) where p = min(31,x+max(w-x,11))
NUMERIC(w,x)	BIGINT	NUMERIC(p,x) where p = min(31,x+max(w-x,19))
NUMERIC(w,x)	NUMERIC(y,z)	NUMERIC(p,s) where p = min(31,max(x,z)+max(w-x,y-z)) and s = max(x,z)
REAL	REAL	REAL
REAL	SMALLINT, INTEGER, BIGINT, DECIMAL, or NUMERIC	DOUBLE
DOUBLE	SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, or DOUBLE	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))

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 is...	And the other operand is...	The data type of the result 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)

If one operand is...	And the other operand is...	The data type of the result is...
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)

The CCSID of the result character string will be derived based on the "Conversion rules for operations that combine strings" on page 95.

Binary string operands

A binary string (BLOB) value is compatible only with another binary string (BLOB) value. The data type of the result is a BLOB. Other data types can be treated as a BLOB data type by using the BLOB scalar function to cast the data type to a BLOB. The length of the result BLOB is the largest length of all the data types.

If one operand is...	And the other operand is...	The data type of the result is...
BLOB(x)	BLOB(y)	BLOB(z) where z = max(x,y)

Datetime operands

A DATE value is compatible with another DATE value or any character string expression that contains a valid string representation of a date. A string representation must not be a CLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is DATE.

A TIME value is compatible with another TIME value, or any character string expression that contains a valid string representation of a time. A string representation must not be a CLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is TIME.

A TIMESTAMP value is compatible with another TIMESTAMP value, or any character string expression that contains a valid string representation of a timestamp. A string representation must not be a CLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is TIMESTAMP.

If one operand is...	And the other operand is...	The data type of the result is...
DATE	DATE, CHAR(y) or VARCHAR(y)	DATE
TIME	TIME, CHAR(y) or VARCHAR(y)	TIME
TIMESTAMP(x)	TIMESTAMP(y)	TIMESTAMP(max(x,y))

If one operand is...	And the other operand is...	The data type of the result is...
TIMESTAMP(x)	CHAR(y) or VARCHAR(y)	TIMESTAMP(x)

Distinct type operands

A user-defined distinct type value is compatible only with another value of the same user-defined distinct type. The data type of the result is the user-defined distinct type.

If one operand is...	And the other operand is...	The data type of the result is...
Distinct type	Distinct type	Distinct type

XML operands

The XML data type is compatible only with another XML data type. The data type of the result is XML.

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 CASE expressions, the IN predicate, and the COALESCE and CONCAT and VALUE scalar functions. 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 X'FFFF', the result is X'FFFF'. [24]
- If one CCSID denotes Unicode data and the other denotes non-Unicode data, the result is the CCSID for Unicode data.
- If one CCSID denotes SBCS data and the other denotes mixed data, the result is the CCSID for mixed data.
- Otherwise, the result CCSID is determined by the following table:

Table 14. 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

G In DB2 for z/OS, derived values are converted before constants and special
G registers.

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 would be determined as follows:

– 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.

24. Both operands must not be a CLOB or DBCLOB.

– 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 COALESCE and CONCAT and VALUE scalar functions 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 X'FFFF'.
- 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 35.

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 35. 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

```
62      -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* is a signed or unsigned number that consists of no more digits than the largest decimal precision and either includes a decimal point or is not within the range of binary integers. For more information on the largest decimal precision, see Chapter 9, “SQL limits,” on page 959. 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 2.

Examples

```
15E1      2.E5      2.2E-1      +5.E+2
```

Decimal floating-point constants

There are no decimal floating-point constants except for the decimal floating-point special values, which are interpreted as DECFLOAT(34).

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, NAN, and SNAN) is used in a context where it could be interpreted as a name, explicitly cast the value to decimal-floating point. For example:

```
CAST('SNAN' 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:

- It is a sequence of characters enclosed between apostrophes. The number of bytes between the apostrophes cannot be greater than 32 672. See Table 60 on page 964 for more information. Two consecutive apostrophes are used to represent one apostrophe within the character string, but these count as one byte when calculating lengths of character constants. Two consecutive apostrophes that are not contained within a string represent an empty string.
- An X followed by a sequence of characters that starts and ends with an apostrophe. The characters between the apostrophes must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 254. See Table 60 on page 964 for more information. 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.

At installations that have mixed data, a character-string constant is classified as mixed data if it includes a DBCS substring. In all other cases, a character-string
G constant is classified as SBCS data. In DB2 for LUW, in a DBCS environment, all
G character string constants are classified as mixed data. The CCSID assigned to the constant is the appropriate default CCSID of the application server at bind time. A mixed data constant can be continued from one line to the next only if the break occurs between single-byte characters.

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 66.

Examples

```
'Peggy'     '14.12.1990'     '32'     'DON''T CHANGE'     ''     X'FFFF'
```

Graphic-string constants

A *graphic-string constant* specifies a varying-length graphic string. The length of the specified string cannot be greater than 16 336. See Table 60 on page 964 for more information.

In EBCDIC environments, the forms of graphic-string constants are :

Graphic String Constant	Empty String	Example
G'<SO>dbcs-string<SI>'	G'<SO><SI>' G'' g'<SO><SI>' g''	G'<SO>元気<SI>'
N'<SO>dbcs-string<SI>'	N'<SO><SI>' N'' n'<SO><SI>' n''	

In ASCII environments, the form of the constant is:

G'*dbcs-string*' or N'*dbcs-string*'

The CCSID assigned to the constant is the appropriate default CCSID of the application server at bind time.

In SQL statements and in host language statements in a source program, graphic strings cannot be continued from one line to another.

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 66. 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.nnnnnnnnnnnn'

The data type of the value is TIMESTAMP(*p*), where *p* is the number of digits of fractional seconds.

Trailing zeros can be truncated or omitted entirely from fractional seconds.

Leading zeros must not be omitted from any part of a standard datetime constant.

Example

```
DATE '2003-09-03'
```

Decimal point

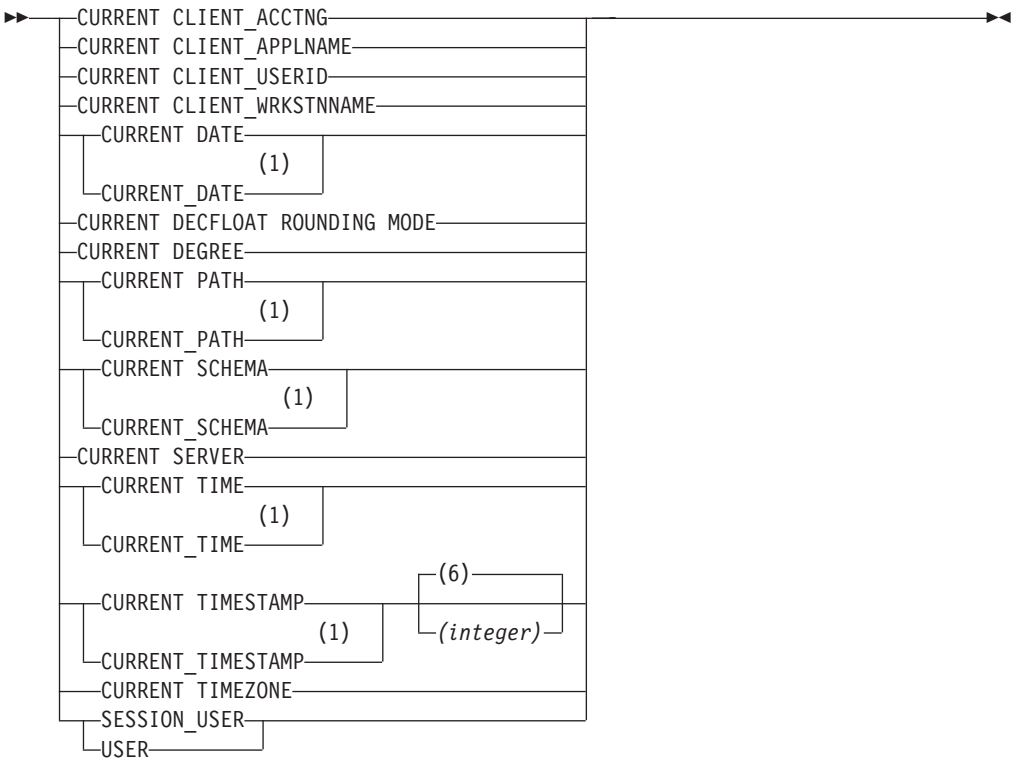
The *default decimal point* can be specified:

- 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, and FLOAT 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 scalar function and the CAST specification)

G Each product provides a product-specific means of explicitly specifying a *default decimal point*.

Special registers

A *special register* is a storage area that is defined for an application process by the 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 2011 Core standard uses the form with the underscore.

The value of these special registers cannot be the null value.

For portability across the platforms, when defining a variable to receive the contents of a special register that contains character data it is recommended that the variable be defined with the maximum length supported by any of the platforms for that special register. For more information on the maximum lengths of the special registers, see Chapter 9, “SQL limits,” on page 959.

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 by using the Set Client Information (sqleseti) API.

Example

Get the current value of the accounting string for the current connection.

```
SELECT CURRENT CLIENT_ACCTNG
  INTO :ACCT_STRING
  FROM SYSIBM.SYSDUMMY1
```

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 by using the Set Client Information (sqleseti) API.

Example

Select the departments that are allowed to use the application being used in the current connection.

```
SELECT DEPT
  FROM DEPT_APPL_MAP
  WHERE APPL_NAME = CURRENT CLIENT_APPLNAME
```

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 by using the Set Client Information (sqleseti) API.

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 by using the Set Client Information (sqleseti) API.

Example

Get the workstation name being used for the current connection.

```
SELECT CURRENT CLIENT_WRKSTNNAME
  INTO :WS_NAME
  FROM SYSIBM.SYSDUMMY1
```

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 or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

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 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_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.

G The initial value of CURRENT DECFLOAT ROUNDING MODE and the method of
G changing the value is product specific.

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 parallelism for the execution of dynamically prepared SQL statements.

G The data type is CHAR(3) in DB2 for z/OS and CHAR(5) in DB2 for LUW and
G DB2 for i.

The value of the CURRENT DEGREE special register is 'ANY' or '1' (padded on the right with blanks). If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is prepared, the execution of the statement will not use parallelism. If the value of CURRENT DEGREE is 'ANY' when an SQL statement is prepared, the execution of that statement can involve parallelism using a degree determined by the database manager.

The initial value of the special register in a user-defined function or procedure is
G inherited from the invoking application. In other contexts the initial value of the
G special register is platform-specific. See the product documentation for details.

The value of the special register can be changed by executing the SET CURRENT DEGREE statement. For more information, see "SET CURRENT DEGREE" on page 879.

Example

The following statement inhibits parallelism.

```
SET CURRENT DEGREE = '1'
```

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 used to resolve unqualified procedure names that are specified as variables in SQL CALL statements (CALL *variable*). The data type is VARCHAR with a length attribute that is the maximum length of a path. For more information, see Chapter 9, "SQL limits," on page 959.

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 name 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.

For information on 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 function, procedure, specific name, type, and variables" on page 52.

The initial value of the special register in a user-defined function or procedure is inherited from the invoking application. In other contexts the initial value of the special register is the system path followed by the USER special register value. For more information on the system path, see "The System Path" in "SET PATH" on page 883.

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 883. 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 the system schemas:

```
SET 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.

For information on when the CURRENT SCHEMA is used to resolve unqualified names in dynamic SQL statements and the effect of its value, see "Qualification of unqualified object names" on page 52.

The initial value of the special register in a user-defined function or procedure is
G inherited from the invoking application. In other contexts the initial value of the
G special register is platform-specific. See the product documentation for details.

The value of the special register can be changed by executing the SET SCHEMA statement. For more information, see "SET SCHEMA" on page 886.

Example

Set the schema for object qualification to 'D123'.

```
SET SCHEMA = 'D123'
```

CURRENT SERVER

The CURRENT SERVER special register specifies a VARCHAR(18) value that
G identifies the current server. In DB2 for z/OS, CURRENT SERVER specifies a
G CHAR(16) value. For more information, see Chapter 9, "SQL limits," on page 959.

The CURRENT SERVER can be changed by the CONNECT (Type 1), CONNECT (Type 2), or SET CONNECTION statements, but only under certain conditions. For more information, see "CONNECT (type 1)" on page 598, "CONNECT (type 2)" on page 602, and "SET CONNECTION" on page 875.

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 or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

Example

Using the CL_SCHED sample 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.

If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading.

If a timestamp with a specific precision is desired, the special register can be referenced as CURRENT TIMESTAMP(*integer*), where *integer* can range from 0 to 12. The default precision is 6.

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 [25] 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). 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. The data type of the register is VARCHAR(128). The initial value is the authorization ID of the user that connected to the current server.

Example

Select all notes from the IN_TRAY sample table that the user placed there.

```
SELECT * FROM IN_TRAY
  WHERE SOURCE = SESSION_USER
```

USER

The USER special register specifies the run-time authorization ID. The data type of the register is VARCHAR(18). See Chapter 9, “SQL limits,” on page 959.

Example

Select all notes from the IN_TRAY sample table that the user placed there.

25. Coordinated Universal Time, formerly known as GMT.

```
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 6, "Queries," on page 463.) 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, 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 statement specifying ON COLUMN, the column name must be qualified.
- Where the column name specifies values of the column, a column name may be qualified.
- In the *assignment-clause* of an UPDATE statement, it may be qualified.
- In the column list of an INSERT statement, a column name 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 110 and "Column name qualifiers in correlated references" on page 112 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. It can also be used merely as a shorter name for a table or view. In the example 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 name or view name, 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
  WHERE EMPLOYEE.PROJECT='ABC'
```

INCORRECT

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 *nonexposed*. 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
    WHERE EMPLOYEE.PROJECT = 'ABC'
```

INCORRECT

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 default schema 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
FROM X.T1, EMPLOYEE T1
```

INCORRECT

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, 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 [26] that might contain the column are

26. In the case of a *joined-table*, each *table-reference* within the *joined-table* is an object table.

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 on avoiding ambiguity between SQL parameters and variables and column names, see "References to SQL parameters and SQL variables" on page 909.

A nested table expression which is preceded by a TABLE keyword will consider *table-reference*s that precede it in the FROM clause as object tables. The *table-reference*s that follow it 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.
- 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.

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.
- 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 TABLE keyword or a table function or nested table expression that is the right operand of a right outer join or full outer 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:

   ```
   SELECT CORPDATA.EMPLOYEE.WORKDEPT
     FROM EMPLOYEE
   ```

   is a valid statement.

2. If the default schema is REGION:

   ```
   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:

   ```
   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 6, "Queries," on page 463 for more information on 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 to retrieve a single value as an expression within a statement is called a scalar fullselect. 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 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 nested table expression that is preceded by the TABLE keyword can also reference columns of the tables that

precede it in the FROM clause. A reference to a column of a table identified at another level is called a *correlated reference*. A reference to a column of a table identified at the same level in a nested table expression that is preceded by the TABLE 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 of a subquery
- Q does not designate an exposed table used in the FROM clause of that subquery
- Q does designate an exposed 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.

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 either built-in global variables or user-defined global variables. 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 116.

transition variable
Transition variables are defined in a trigger and refer to either the old or new values of columns of the subject table of a trigger. For more information about how to refer to transition variables see "CREATE TRIGGER" on page 718.

SQL variable
SQL variables are defined by an SQL compound statement in an SQL function or SQL procedure. For more information about SQL variables, see "References to SQL parameters and SQL variables" on page 909.

SQL parameter
SQL parameters are defined in an CREATE FUNCTION (SQL Scalar) or CREATE PROCEDURE (SQL) statement. For more information about SQL parameters, see "References to SQL parameters and SQL variables" on page 909.

parameter marker
Parameter markers are specified in an SQL statement that is dynamically prepared instead of host variables. For more information about parameter markers, see "Notes" on page 837 in the PREPARE statement.

Global variables

Global variables are named memory variables that you can access and modify through SQL statements.

DB2 SQL supports the following types of global variables:

Built-in global variable
A built-in global variable is part of the database management system, and is available to any SQL statement that runs on the database manager. Built-in global variables reside in the SYSIBM schema. For a list of the built-in global variables and information on these variables, see Chapter 4, "Built-in global variables," on page 181.

User-defined global variable
A user-defined global variable enables you to share relational data between SQL statements without the need for application logic to support this data transfer.

A user-defined global variable is associated with a specific session and contains a value that is specific to that session. A user-defined session global variable is available to any active SQL statement running against the database on which the variable was defined. A user-defined global variable

can be associated with more than one session, but its value will be specific to each session. User-defined global variables are defined in the system catalog.

You can control access to global variables through the GRANT (Global Variable Privileges) and REVOKE (Global Variable Privileges) statements.

Global variable names are qualified names. If an unqualified global variable name is the main object of an ALTER, CREATE, COMMENT, DROP, GRANT, or REVOKE statement, the name is implicitly qualified using the same rules as for qualifying unqualified table names. Otherwise, when a global variable is referenced without the schema name, the SQL path is used for name resolution.

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.

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 almost 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)
- Indexes

Authorizations: 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.
- Database administrator authority

The value of a global variable can be changed using the FETCH, 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 or a C[27], Java, or REXX 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 118.

A *host variable* in an SQL statement must identify a host variable described in the program according to the rules for declaring host variables.

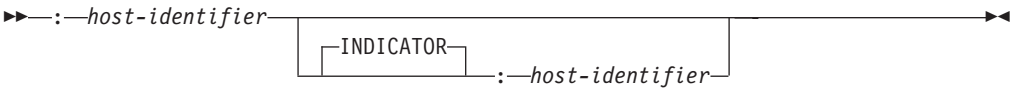
All host variables used in an SQL statement must be declared in an SQL declare section in all host languages other than Java and REXX. Variables do not have to be declared in REXX. In Java, variables must be declared, but an SQL declare section is not necessary or allowed. No variables may be declared 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:

- Chapter 15, "Coding SQL statements in C applications," on page 1063
- Chapter 16, "Coding SQL statements in COBOL applications," on page 1081
- Chapter 17, "Coding SQL statements in Java applications," on page 1099
- Chapter 18, "Coding SQL statements in REXX applications," on page 1117

A variable in the INTO clause of a FETCH, a SELECT INTO, or a VALUES INTO statement identifies a host variable to which a column value is assigned. A host variable in a CALL 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 be a 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 the null value. A negative value of the indicator variable specifies the null value.
- Indicate that a numeric conversion error (such as a divide by 0 or overflow) has occurred. [28]

27. In this book, whenever the C language is referenced, the information also applies to C++.

28. In DB2 for LUW, the database configuration parameter dft_sqlmathwarn must be set to yes for this behavior to be supported.

- Indicate that a character could not be converted.
- Record the original length of a truncated string, if the string is not a LOB.
- Record the seconds portion of a time if the time is truncated on assignment to a host variable.

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 if 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
- -2 if the null value was returned due to a numeric conversion error (such as divide by 0 or overflow) or a character conversion error. [29]

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 a time, V2 is set to the number of seconds.

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, do not use this form 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.

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.

G Extended indicator variables are only enabled if requested. Otherwise all indicator
G variables are normal indicator variables. The method for enabling extended indicator variables is product-specific 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.

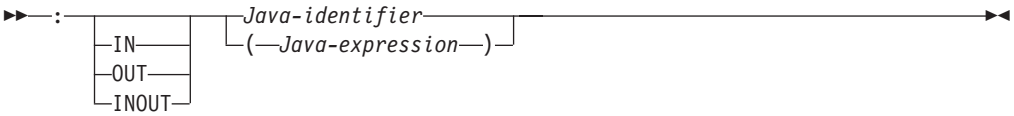
29. Note that although a -2 null value can be returned for conversion errors, the result column itself is not considered nullable unless an argument of the expression, scalar function, the column is nullable.

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 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 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. In this case the host variable value 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 can not 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 on Java variables, see "Using host variables and expressions in Java" on page 1104.

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 a host variable, SQL variable, or SQL parameter 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 host variable, SQL variable, or SQL parameter would be found if the statement string were a static SQL statement. The following examples shows a static SQL statement 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 "PREPARE" on page 833.

References to LOB variables

Regular LOB variables and LOB file reference variables can be defined in all host languages other than REXX. LOB locator variables can be defined in the following host languages:

- C
- COBOL

Where LOBs 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 value and the performance benefit of delaying the transfer of data from the server is not required, a LOB locator is not needed. However, it is often not acceptable to store an entire LOB value in temporary storage due to host language restrictions, storage restrictions, or performance requirements. When storing an entire LOB value at one time is not acceptable, a LOB value can be referenced using a LOB locator and portions of the LOB value can be accessed.

References to LOB locator variables

A LOB *locator variable* is a variable that contains the locator representing a LOB value on the server, which can be defined in the following host languages:

- C
- COBOL

See "Manipulating large objects with locators" on page 63 for information on how locators can be used to manipulate LOB values.

A locator variable in an SQL statement must identify a LOB locator variable described in the program according to the rules for declaring locator variables. For example, in C:

```
static volatile SQL TYPE IS CLOB_LOCATOR *loc1;
```

Like all other variables, a LOB locator variable can have an associated indicator variable. Indicator variables for LOB 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 locator is null, the value of the referenced LOB 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 locators that were acquired by the transaction are released.

It is the application programmer's responsibility to guarantee that any LOB locator is only used in SQL statements that are executed at the same server that originally generated the LOB locator. For example, assume that a LOB locator is returned

from one 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 server unpredictable results will occur.

References to LOB file reference variables

A LOB *file reference variable* is used for direct file input and output for a LOB. A LOB *file reference variable* can be defined in the following host languages:

- C
- COBOL

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 locator represents, rather than contains, the LOB 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 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.

For more information on file reference variables, see the product references.

References to XML variables

XML variables are defined in host languages as a type of LOB variable.

XML host variables can be declared as another host variable type. This allows XML data to be declared in host languages that do not support XML data as a native data type.

XML host variables can be CLOB, DBCLOB, or BLOB data. 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 C structure or COBOL group, that is referred to 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 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 referred to 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 the variables within the structure S1, the statement:

```
EXEC SQL FETCH CURSOR1 INTO :S1;
```

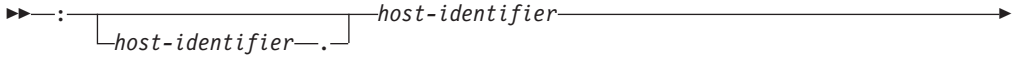
is equivalent to:

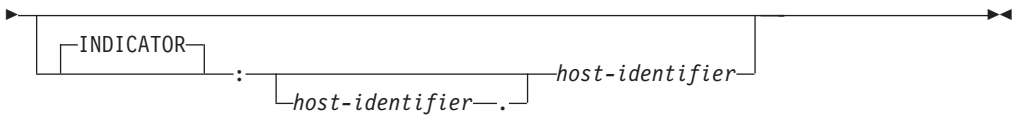
```
EXEC SQL FETCH CURSOR1 INTO :V1, :V2, :V3;
```

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 variable is not specified, no variable of the host structure has an indicator variable.

In addition to structure references, individual host variables in a host structure or indicator variables in an indicator array can be referred to 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 structure and the second host identifier must name a host variable within that 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
```

Functions

A *function* is an operation denoted by a function name followed by zero 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 on these functions, see Chapter 5, "Built-in functions," on page 187.

  G The *built-in functions* are in a product-specific schema.
- *User-defined functions* are functions that are created using the CREATE FUNCTION statement and registered to the database manager in the catalog. For more information, see "CREATE FUNCTION" on page 606. 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 an *SQL*, *external*, or *sourced* function. An SQL function is defined to the database using only an SQL RETURN statement. An external function is defined to the database with a reference to an external 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.
- *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 user-defined functions for distinct types reside in the same schema as the distinct type for which they were created. For more information about the functions that are generated for a distinct type, see "CREATE TYPE (distinct)" on page 734.

Another way to classify functions is as aggregate, scalar, or table functions, depending on the input data values and result values.

- 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.

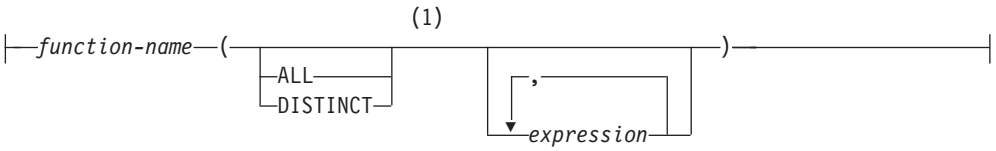
- A *table function* returns a table for the set of arguments it receives. Each argument is a single value. A table function can be referenced only in the FROM clause of a subselect. A table function can be defined as an external function, but a table function cannot be a sourced function.

  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 function conforms to the following syntax:[30]

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.

In the above syntax, *expression* cannot include an aggregate function. See "Expressions" on page 130 for other rules for *expression*.

When the user-defined 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 scalar function or a user-defined aggregate function is 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 77.

Additionally, a character FOR BIT DATA argument cannot be passed as input for a parameter that is not defined as character FOR BIT DATA. Likewise, a character argument that is not FOR BIT DATA cannot be passed as input for a parameter that is defined as character FOR BIT DATA.

Table functions can be referenced only in the FROM clause of a subselect. For more details on referencing a table function, see "table-reference" on page 470.

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

30. A few functions allow keywords instead of a comma separated list 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 POSITION function uses a keyword.

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 52), 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 SQL path 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" on page 126). 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 the system portion of the SQL path. For more information on the system portion of the SQL path, see "SQL path" on page 52. 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 statement, function resolution occurs at the time the view is created. If another function with the same name is subsequently created, the view is not affected, even if the new function is a better fit than the one chosen at the time the view was created. In CREATE FUNCTION, CREATE MASK, CREATE PERMISSION, CREATE PROCEDURE, and CREATE TRIGGER statements, the effect of subsequently creating a function with the same name that is a better fit is product-specific.

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, CCSID, etc. are ignored. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), and DECIMAL(4,3).

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 above 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 candidate functions, that function is the best fit. The precedence list for the promotion of data types in "Promotion of data types" on page 72 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 124. 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 the application invokes the 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 124.
- 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 is returned. 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 is returned 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 storage assignment rules (see “Assignments and comparisons” on page 77). 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.

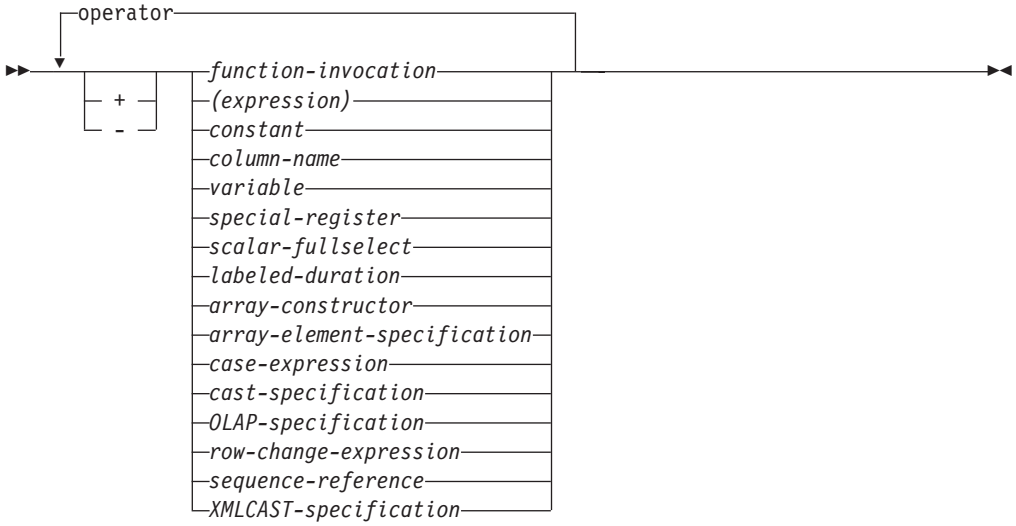
- The function referenced in an SQL statement requires a scalar or aggregate function, but the function selected by the function resolution step is a table function.

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 114.
- *scalar-fullselect*. For information about authorization considerations, see Chapter 6, "Queries," on page 463.
- *sequence-reference*. The authorization to reference the sequence. For information about authorization considerations, see Sequence authorization.
- *function-invocation*. The authorization to execute the function. For information about authorization considerations, see "Function invocation" on page 124.
- *array-element-specification*. The authorization to reference an array type. For information about authorization considerations, see "GRANT (type privileges)" on page 804.
- *cast-specification*. The authorization to reference a user-defined type. For information about authorization considerations, see "CAST specification" on page 149.



operator:

Without operators

If no operators are used, the result of the expression is the specified value.

Examples

```
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 is the null value, the result of the expression is the null value. Arithmetic operators must not be applied to character strings. For example, USER+2 is invalid.

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, and division, respectively. If the value of the second operand of division is zero, then an error is returned.

In COBOL, blanks must precede and follow a minus sign to avoid any ambiguity with COBOL host variable names (which allow use of a minus sign).

Operands with a string data type are cast to numeric. The string must contain a valid string representation of a number. The data type the string is cast to is product-specific.

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.

Integer and decimal operands

If one operand is an integer 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 *p* and scale 0. *p* is 19 for a big integer, 11 for a large integer, and 5 for a small integer. However, in the case of an integer constant, *p* is product-specific.

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 as if a temporary copy of the operand with the smaller scale is made by extending it with trailing zeros so that its fractional part has the same scale as the longer operand.

Unless specified otherwise, all functions and operations that accept decimal numbers allow a precision of up to 31 digits. The result of a decimal operation must not have a precision greater than 31.

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.

Addition and subtraction: The scale of the result of addition and subtraction is max (*s*,*s'*). The precision is min(31,max(*p*-*s*,*p'*-*s'*) +max(*s*,*s'*)+1). [31]

Multiplication: The precision of the result of multiplication is min (31,*p*+*p'*) and the scale is min(31,*s*+*s'*). In DB2 for z/OS, special rules apply if both p and p' are greater than 15. See the product reference for further information.

Division: The precision of the result of division is 31. The scale is *31*-*p*+*s*-*s*'. The scale must not be negative. In DB2 for z/OS, the scale is different and special rules apply when p' is greater than 15. See the product reference for further information.

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 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.

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 database manager (for example, the database manager may decide what degree of parallelism to use and what access plan to use), an application that uses floating-point operands should not depend on the results being precisely the same each time an SQL statement is executed.

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.

31. For DB2 for z/OS, the formulas used in this book are those that apply when the DEC31 option is in effect or the precision of an operand is greater than 15.

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 103.

  If the value of the adjusted exponent of the result is less than E_{min}, then a subnormal warning is returned. 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 is 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.[32]

  If the value of the adjusted exponent of the result is larger than E_{max} then an overflow warning is returned.[32] In this case, the result may be infinite. It will have the same sign as the theoretical result.

G DB2 for LUW does not return subnormal warnings.

- 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: [32]
  - Add +infinity to -infinity during an addition or subtraction operation
  - Multiply 0 or -0 by +infinity or -infinity
  - 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.

32. In DB2 for i the warning is only returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

- 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 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.[33]

33. Using the vertical bar (|) character might inhibit code portability between DB2 relational database products. Use the CONCAT operator in place of the || operator. On the other hand, if conformance to the SQL 2011 Core standard is of primary importance, use the || operator).

The operands of concatenation must be compatible strings or numbers that are not distinct types. When any operand is a non-string value, it is implicitly cast to VARCHAR. Note that a binary string cannot be concatenated with a character string, including character strings 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 15. 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)* 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)	CHAR(y)	CHAR(z)** z = x+y and z must not be greater than the maximum length of CHAR
BLOB(x)	BLOB(y)	BLOB(z) where z = MIN(x + y, maximum length of a BLOB)
Note: * Character strings are only allowed when the other operand is a graphic string if the graphic string is Unicode. ** In EBCDIC environments, if either operand is mixed data, the resulting data type is VARCHAR(z) or VARGRAPHIC(z). In DB2 for z/OS and DB2 for LUW, if z evaluates to greater than the maximum length of a CHAR column, then VARCHAR(z) where z = x + y. *** In DB2 for LUW, if z evaluates to greater than 2000, a LONG VARGRAPHIC is returned. **** In DB2 for LUW, if z evaluates to greater than 4000, a LONG VARCHAR is returned.		

The encoding scheme of the result is summarized in the following table:

Table 16. 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 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 both operands are strings, the sum of their lengths must not exceed the maximum length of the resulting data type. See Table 60 on page 964 for more information.

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 EBCDIC 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, while the second operand is a character string beginning with a "shift-out" character, these two bytes are eliminated from the result.

The length of the result is the sum of the lengths of the operands, unless redundant shift codes are eliminated, in which case the 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 95. 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.

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 500 for more information.

A scalar fullselect cannot be used in the following instances:

- A CHECK constraint in CREATE TABLE and ALTER TABLE statements
- A view definition that has a WITH CHECK OPTION
- A grouping expression

- A partitioning expression in an OLAP specification
- An argument in a CALL statement for an input parameter
- An argument to an aggregate function, other than the XML-expression argument of the XMLAGG function
- An ORDER BY clause
- A join-condition of the ON clause for INNER and OUTER JOINs
- A CREATE FUNCTION (SQL) statement

If the scalar fullselect is a subselect, it is also referred to as a scalar subselect. See "subselect" on page 464 for more information.

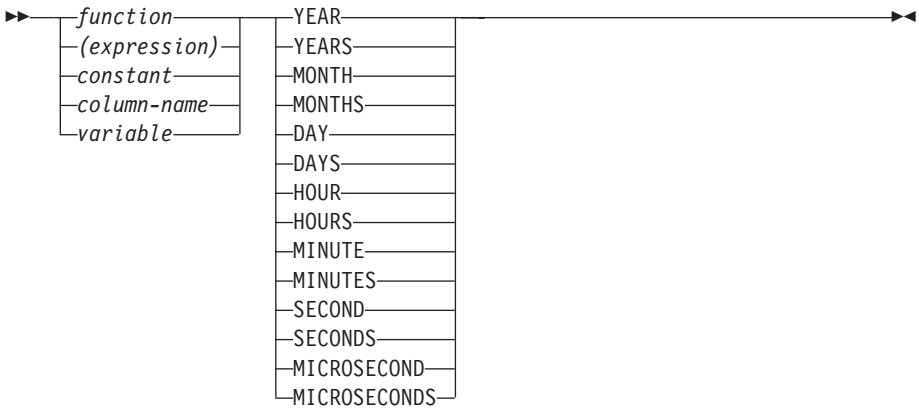
Datetime operands and durations

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

The form a labeled duration is as follows:

labeled-duration:



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 duration keywords. 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, 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 - BIRTHDATE, 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 fractional seconds, expressed as a DECIMAL(14+*s*,*s*) number, where *s* is the number of digits of fractional seconds ranging from 0 to 12.. To be properly interpreted, the number must have the format *yyyymmddhhmmss.zzzzzzzzzzzz*, where *yyyy, mm, dd, hh, mm, ss, and zzzzzzzzzzzz* represent, respectively, the number of years, months, days, hours, minutes, seconds, and fractional seconds. The result of subtracting one timestamp value from another is a timestamp duration with scale that matches the maximum timestamp precision of the timestamp operands.

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 string representation of a timestamp, it is implicitly converted to a timestamp with the same precision as the first operand.

- If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp. If the first operand is a string representation of a timestamp, it is implicitly converted to a timestamp with the same precision as the second operand.
- 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, and a warning indicator in the SQLCA is set 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 a warning indicator in the SQLCA is set 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 or subtracting a 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

When adding durations to dates, adding one month to a given date gives the same date one month later *unless* that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

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 fractional seconds between the two timestamps. The data type of the result is DECIMAL(14+*s*,*s*), where *s* is the maximum timestamp precision of TS1 and TS2. 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 SECOND(TS2,s) <= SECOND(TS1,s)
    then SECOND(RESULT,s) = SECOND(TS1,s) - SECOND(TS2,s).

If SECOND(TS2,s) > SECOND(TS1,s)
then SECOND(RESULT,s) = 60 + SECOND(TS1,s) - SECOND(TS2,s,)
and MINUTE(TS2) is incremented by 1.

The 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. The precision of the result timestamp matches the precision of the timestamp operand. 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. Fractional seconds overflow into seconds. Thus, subtracting a duration, X, from a timestamp, TIMESTAMP1, where X is a DECIMAL(14+*s*,*s*) number is equivalent to the expression:

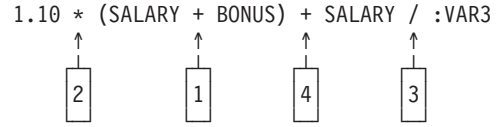
TIMESTAMP1 - YEAR(X) YEARS - MONTH(X) MONTHS - DAY(X) DAYS
- HOUR(X) HOURS - MINUTE(X) MINUTES
- SECOND(X,s) SECONDS

When subtracting a duration with non-zero scale, a labeled duration of MICROSECOND or MICROSECONDS, or a labeled duration of SECOND or SECONDS with a value that includes fractions of a second, the subtraction is performed as if the timestamp value has up to 12 fractional second digits. The resulting value is assigned to a timestamp value with the timestamp precision of the timestamp operand which could result in truncation of fractional second digits.

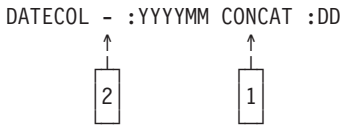
Precedence of operations

Expressions within parentheses are evaluated first. When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and division, and multiplication, division, and concatenation are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

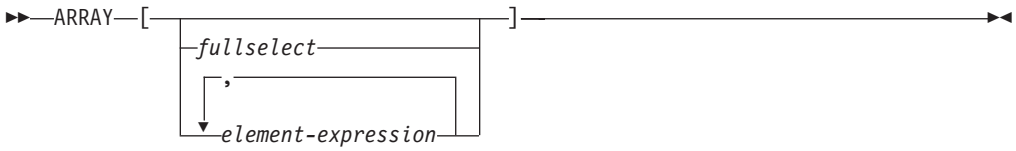
Example 1: In this example, the first operation is the addition in (SALARY + BONUS) because it is within parentheses. The second operation is multiplication because it is at a higher precedence level than the second addition operator and it is to the left of the division operator. The third operation 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. The attributes of the base type of the array is the same as the data type of the result column of the fullselect.

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 array index 1. The second *element-expression* is assigned to the array element with array index 2, and so on. All element expressions must have compatible data types. The attributes of the base type of the array is determined by all the operands as explained in "Rules for result data types" on page 91.

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 an *assignment-statement*.

Examples

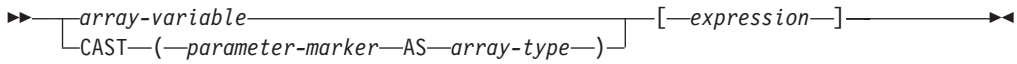
- Set the array variable RECENT_CALLS of array type PHONENUMBERS to an array of fixed numbers.

```
SET RECENT_CALLS = ARRAY[9055553907, 4165554213, 4085553678]
```

- Set the array variable DEPT_PHONES of array type PHONENUMBERS to an array of phone numbers retrieved from the DEPARTMENT_INFO table.

```
SET DEPT_PHONES = ARRAY[SELECT DECIMAL(AREA_CODE CONCAT '555' CONCAT EXTENSION,16)
                        FROM DEPARTMENT_INFO
                        WHERE DEPTID = 624]
```

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 or SQL function.

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 array index of the element that is to be extracted from the array. The array index must return a value that is an exact numeric with zero scale or DECFLOAT. Its value must be between 1 and the cardinality of the array.

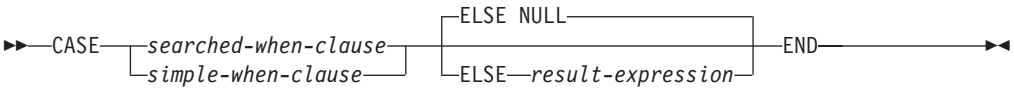
expression must not be:

- An expression that references the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

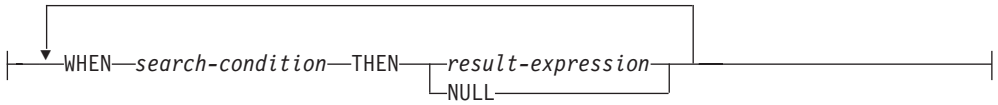
The data type of the result is the data type specified for the array on the CREATE TYPE (Array) statement.

If *expression* is null or the array is null, the null value is returned.

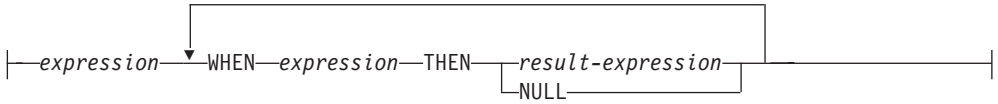
CASE expressions



searched-when-clause:



simple-when-clause:



CASE expressions allow an expression to be selected based on the evaluation of one or more conditions. 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 if 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.
- must not be a CLOB, DBCLOB or BLOB or a character string with a maximum length greater than 254 or a graphic string with a maximum length greater than 127.
- must not include a function that is non-deterministic or has an external action.

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 91.

search-condition
Specifies a condition that is true, false, or unknown about a row or group of table data.

If the CASE expression is in a select list, a VALUES clause, or an IN predicate, the *search-condition* cannot contain a quantified predicate. The *search-condition* cannot contain an IN predicate or an EXISTS predicate.

If a CASE expression is used in a select list, the SET clause of an UPDATE or MERGE statement, or the VALUES clause of an INSERT or MERGE statement, and if simple-when-clause or searched-when-clause references a column for which column access control is activated, the masked value is used instead of the column value.

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 17. 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
```

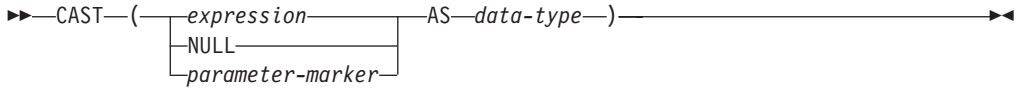
- Another interesting example of CASE statement usage is in protecting from division by 0 errors. For example, the following code finds the employees who earn more than 25% of their income from commission, but who are not fully paid on commission:

```
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:

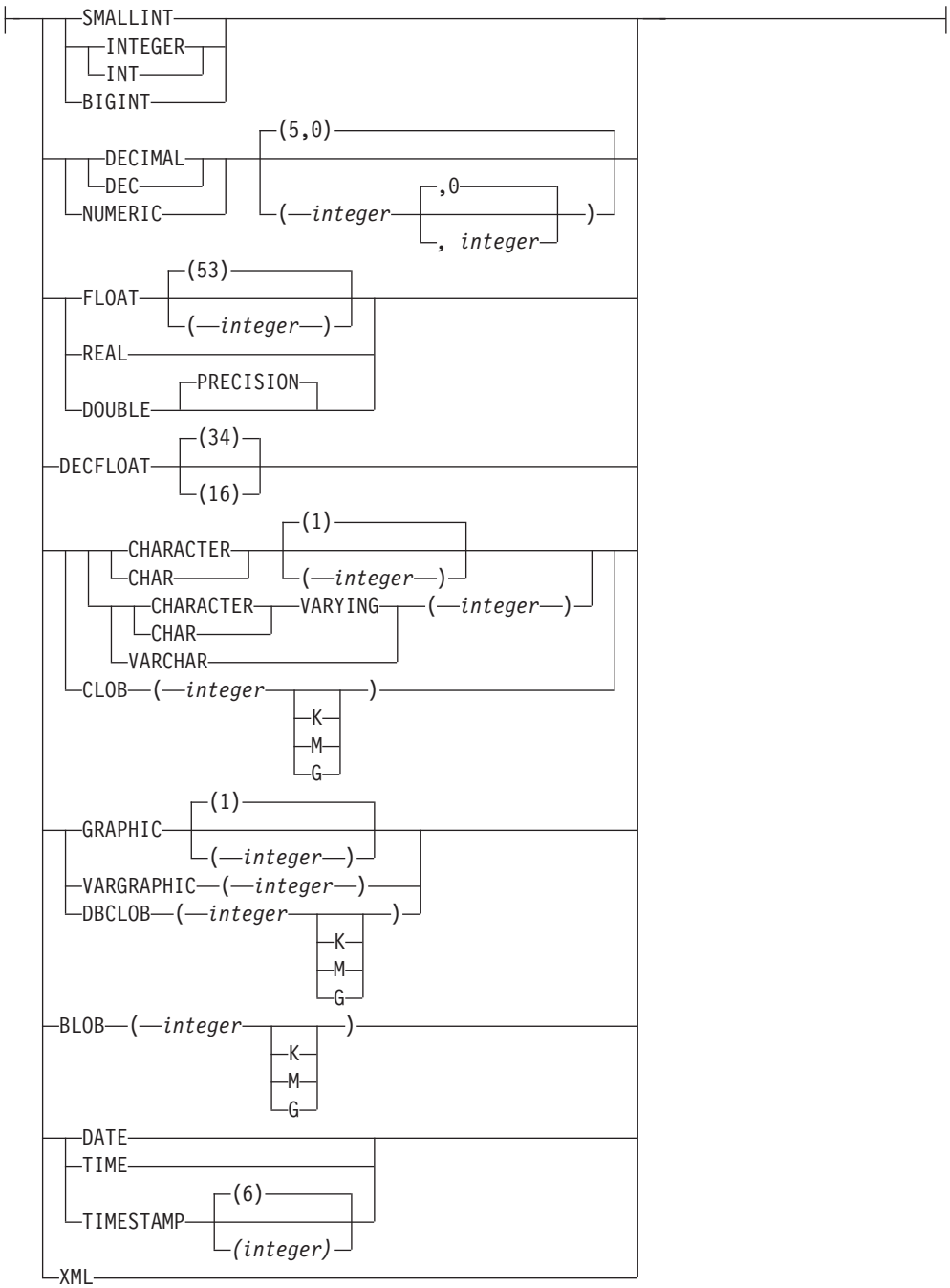
```
SELECT LASTNAME,
 CASE
 WHEN LASTNAME = 'Haas' THEN 'President'
 ...
 ELSE 'Unknown'
 END
FROM EMPLOYEE

SELECT LASTNAME,
 CASE LASTNAME
 WHEN 'Haas' THEN 'President'
 ...
 ELSE 'Unknown'
 END
FROM EMPLOYEE
```

CAST specification



data-type:

built-in-type:



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 EXECUTE on the generated user-defined functions for the distinct type. If the data type of the second 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 74, 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.

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 77). Such a parameter marker is called a *typed parameter marker*. Typed parameter markers are 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 function, procedure, specific name, type, and variables" on page 52. For a description of *data-type*, see "CREATE TABLE" on page 688. (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 9 on page 75 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.

For information on which casts between data types are supported and the rules for casting to a data type see "Casting between data types" on page 74.

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
```

- Assume that two distinct types exist. T_AGE is sourced on SMALLINT and is the data type for the AGE column in the PERSONNEL table. R_YEAR is 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.

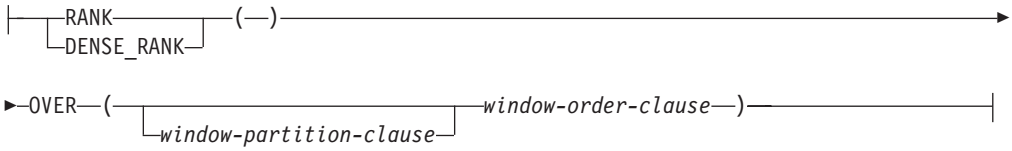
The second parameter marker is a typed parameter marker that is cast to distinct type T_AGE. An explicit CAST specification is required in this case because the parameter marker value is compared to the distinct type.

OLAP specification

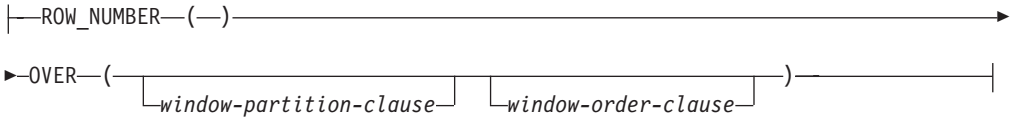
On-Line Analytical Processing (OLAP) specifications provide the ability to return ranking and row numbering as a scalar value in a query result.

OLAP-specification:

ordered-OLAP-specification:

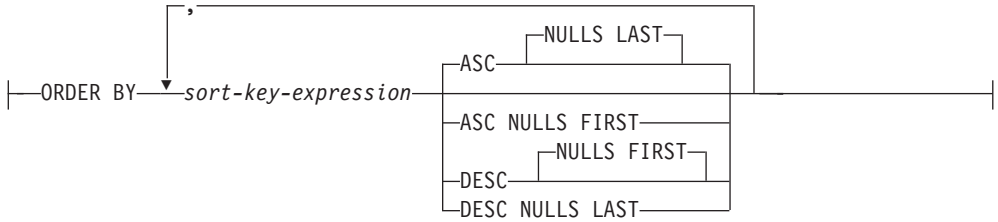


numbering-specification:



window-partition-clause:

window-order-clause:



An OLAP specification can be included in an expression in a *select-clause* 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, 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-clause* .

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,...)*
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.

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.

Partitioning and ordering are performed in accordance with the comparison rules described in "Assignments and comparisons" on page 77.

If a column that is referenced in the partitioning expression or the sort key expression of the OLAP specification is defined to have a column mask, the column mask is not applied.

Notes

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
```

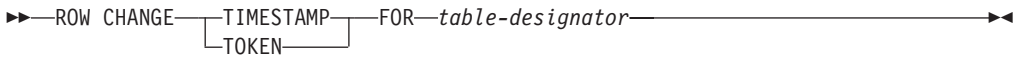
- List the top five wage earners:

```
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 111. 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. If the *table-designator* is a view or nested table expression, it must be deleteable. The view or nested table expression must contain only one base table in its outer subselect. The *table-designator* must not identify a view or a nested table expression that contains a subquery that is materialized.

The result can be the null value. These expressions are not deterministic.

Examples

- Find all rows that have been changed in the last day:

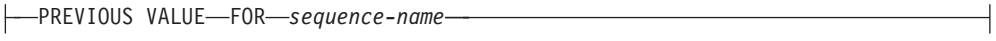
```
SELECT *
  FROM ORDERS
  WHERE ROW CHANGE TIMESTAMP FOR ORDERS > CURRENT TIMSTAMP - 24 HOURS
```

Sequence reference

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, or
  - Ownership of the sequence
- Database administrator authority

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 an EXCEPT keyword.
- 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)

Restrictions on the use of NEXT VALUE and PREVIOUS VALUE: NEXT VALUE and PREVIOUS VALUE expressions cannot be specified in the following places:

- Join condition of a full outer join
- Within a materialized query table definition in a CREATE TABLE or ALTER TABLE statement
- Within a CHECK constraint
- Within a view definition
- 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 operators
- Join condition of a join
- Nested table expression
- Parameter list of a table function
- *select-clause* of the subselect 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 can be specified in place of 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:

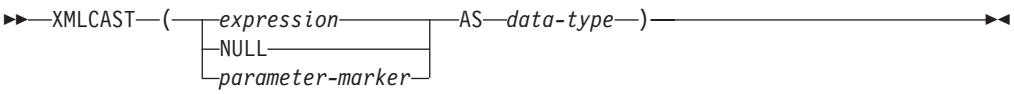
```
CREATE SEQUENCE ORDER_SEQ
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 24
```

  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
```

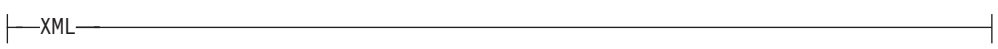
XMLCAST specification

The XMLCAST specification returns the cast operand (the first operand) cast to the XML *data-type*.



data-type:

built-in-type:



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.

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.

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.

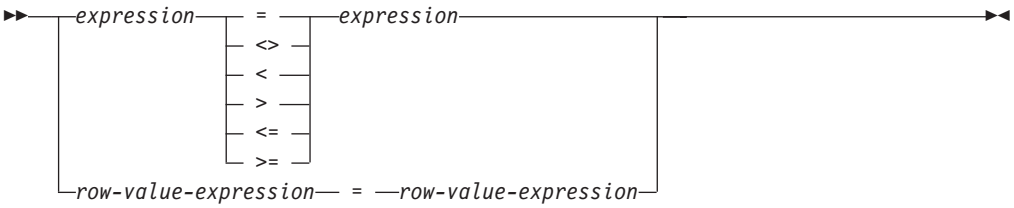
The following rules apply to all types of predicates:

- Predicates are evaluated after the expressions that are operands of the predicate.
- All values specified in the same predicate must be compatible.
- An expression used in a basic, quantified, IN, or BETWEEN predicate must not result in a character string with a length attribute greater than 4 000, a graphic string with a length attribute greater than 2 000, or a LOB string of any size.
- The value of a variable may be null.
- The CCSID conversion of operands of predicates involving two or more operands are done according to "Conversion rules for comparison" on page 88.

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.

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 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.

The result of the predicate is:

- True if all pairs of corresponding value expressions evaluate to true.
- 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).

For values *x* and *y*:

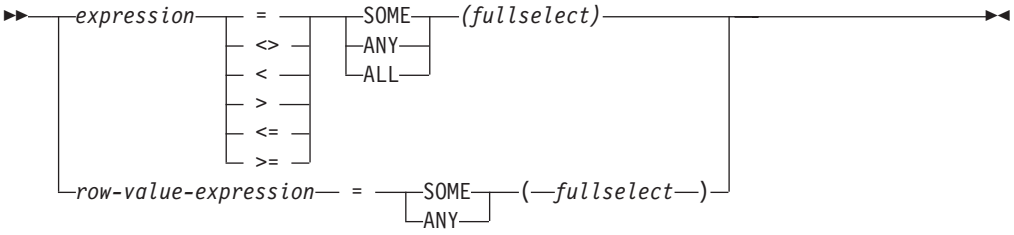
Predicate	**Is true if and only if...**
x **=** *y*	*x* is equal to *y*
*x***<>** *y*	*x* is not equal to *y*
x **<** *y*	*x* is less than *y*
x **>** *y*	*x* is greater than *y*
*x***>=** *y*	*x* is greater than or equal to *y*
*x***<=** *y*	*x* is less than or equal to *y*

Examples

```
EMPNO = '528671'

PRTSTAFF <> :VAR1
```

```
SALARY + BONUS + COMM < 20000

SALARY > (SELECT AVG(SALARY)
           FROM EMPLOYEE)
```

Quantified predicate



A *quantified predicate* compares a value or values with a set of values.

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-expression* 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 list 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.

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)
```

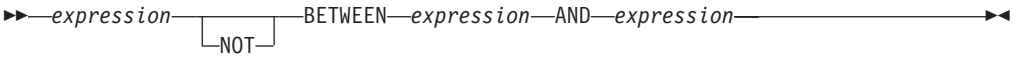
Results in 1,2,3,4, and null. The fullselect returns no values. Thus, the predicate is true for all rows in TBLA.

Example 5

```
SELECT * FROM TBLA WHERE COLA > ANY(SELECT COLB FROM TBLB WHERE COLB<0)
```

Results in the empty result table. The fullselect returns no values. Thus, the predicate is false for all rows in TBLA.

BETWEEN predicate



The BETWEEN predicate compares a value with a range of values.

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)
```

that is,

```
value1 < value2 OR value1 > value3
```

G If the operands of the BETWEEN predicate are strings with different CCSIDs,
G product-specific rules are used to determine which operands are converted.

Given a mixture of datetime values and string representations of datetime values,
G all operands are converted to the data type of the datetime operand. For some
G cases, DB2 for z/OS will not convert character operands to the data type of the
G datetime operand. Use the DATE, TIME or TIMESTAMP scalar function to
G explicitly convert character operands to the appropriate datetime data type.

Examples

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000

SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

EXISTS predicate

The EXISTS predicate tests for the existence of certain rows. The fullselect may specify any number of columns, and

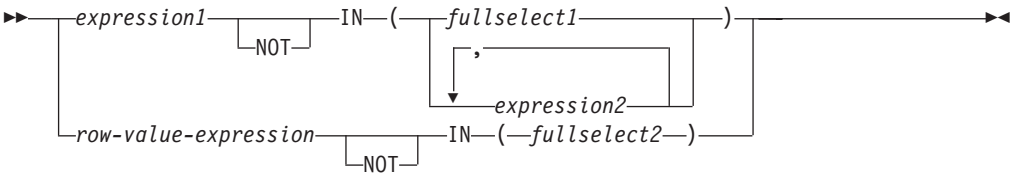
- The result is true only if the number of rows specified by the fullselect is not zero
- The result is false only if the number of rows specified by the fullselect is zero
- The result cannot be unknown.

Any values that may be 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 *fullselect1* or *expression2* must be compatible. If *expression2* is a single host variable, the host variable can identify a structure. 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 the *fullselect2*.
- The result table of the *fullselect2* must have the same number of columns as the *row-value-expression*. The data type of each expression in *row-value-expression* and the data type 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 *fullselect2* is equal to the *row-value-expression*.
  - False if the result of *fullselect2* is empty or if no row returned from *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**)**

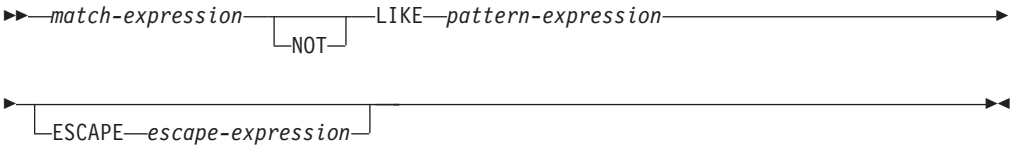
If the operands of the IN predicate have different data types or attributes, the rules used to determine the data type for evaluation of the IN predicate are those for UNION, UNION ALL, EXCEPT, and INTERSECT. For a description, see “Rules for result data types” on page 91.

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 95.

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 a string. The values for *match-expression*, *pattern-expression*, and *escape-expression* must either all be binary strings or none can be binary strings.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source data type.

With character strings, the terms *character*, *percent sign*, and *underscore* in the following description 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. The maximum length of pattern-expression must not be larger than 4000 bytes.

The expression can be specified by any one of:

- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- An expression concatenating any of the above

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.

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 *y* is an empty string and *x* 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 the CCSID of either the pattern value or the escape value is different than that of the column, that value is converted to adhere to the CCSID of the column before the predicate is applied.

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 (either SBCS or DBCS) refers to any number of characters of any type, either SBCS or DBCS.

In EBCDIC environments, any redundant shifts in either the column values or the pattern are ignored. [34]

If the pattern is improperly formed mixed data, the result is unpredictable.

Unicode data: For Unicode, the special characters in the pattern are interpreted as follows:

- An SBCS or DBCS underscore refers to one character (either SBCS or MBCS)
- A percent sign (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

When the LIKE predicate is used with Unicode data, the Unicode percent sign and underscore use the code points indicated in the following table:

Table 18. Code Points for Unicode Percent Sign and Underscore

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'

34. In DB2 for i, redundant shifts are normally ignored, but the IGNORE_LIKE_REDUNDANT_SHIFTS option must be specified to ensure redundant shifts are always ignored.

Table 18. Code Points for Unicode Percent Sign and Underscore (continued)

Character	UTF-8	UTF-16 or UCS-2
Full-width _	X'EFBCBF'	X'FF3F'

The full-width or half-width % matches zero or more characters. The full-width or half-width _ character matches exactly one character. (For ASCII or EBCDIC data, a full-width _ character matches one DBCS character.)

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 the 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 the search was intended to search only for the values that start with 'WYSE', then assign the value 'WSYE%%%%%%' to the variable.

ESCAPE *escape-expression*

The ESCAPE clause allows the definition of patterns intended to match values that contain the actual percent and underscore characters.

The expression can be specified by any one of:

- A constant
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)

The following rules also apply to the use of the ESCAPE clause and *escape-expression*:

- The result of *escape-expression* must be one SBCS or DBCS character or a binary string that contains exactly 1 byte. [35]
- If the value of the ESCAPE *variable* is null, the result of the predicate is unknown.
- The *pattern-expression* forming the pattern 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.
- If the column is mixed data, the ESCAPE clause is not allowed.

The following example shows the effect of successive occurrences of the escape character, which in this case is the plus sign (+).

When the pattern string is...	The actual pattern is...
+%	A percent sign

35. Except in C, where a NUL-terminated character or graphic string variable of length 2 can be used.

When the pattern string is...	The actual pattern is...
++%	A plus sign followed by zero or more arbitrary characters
+++%	A plus sign followed by a percent sign

Examples

Example 1: 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.

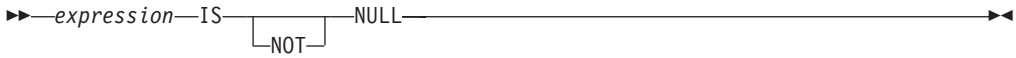
Example 4: In the following table of EBCDIC examples, assume COL1 is mixed data. The table shows the results when the predicates in the first column are evaluated using the COL1 values from the second column:

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaa SO AB%C SI'	'aaa SO ABDZC SI'	True
	'aaa SO AB SI dzx SO C SI'	True
	Empty string	False
WHERE COL1 LIKE 'aaa SO AB SI % SO C SI'	'aaa SO ABDZC SI'	True
	'aaa SO AB SI dzx SO C SI'	True
	Empty string	False
WHERE COL1 LIKE 'SO SI'	'aaa SO ABDZC SI'	False
	'aaa SO AB SI dzx SO C SI'	False
	Empty string	True
WHERE COL1 LIKE 'SO % SI'	'aaa SO ABDZC SI'	True
	'aaa SO AB SI dzx SO C SI'	True
	Empty string	False
WHERE COL1 LIKE 'SO _____ SI %'	'aaa SO ABDZC SI'	True
	'aaa SO AB SI dzx SO C SI'	True
	Empty string	False
WHERE COL1 LIKE 'SO _____ SI'	'aaa SO ABDZC SI'	False
	'aaa SO AB SI dzx SO C SI'	False
	Empty string	False

Example 5: In the following table of ASCII 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 'aaaAB%C'	'aaaABDZC'	True
WHERE COL1 LIKE 'aaaAB%C'	'aaa AB dzx C'	True

NULL predicate



The NULL predicate tests for null values.

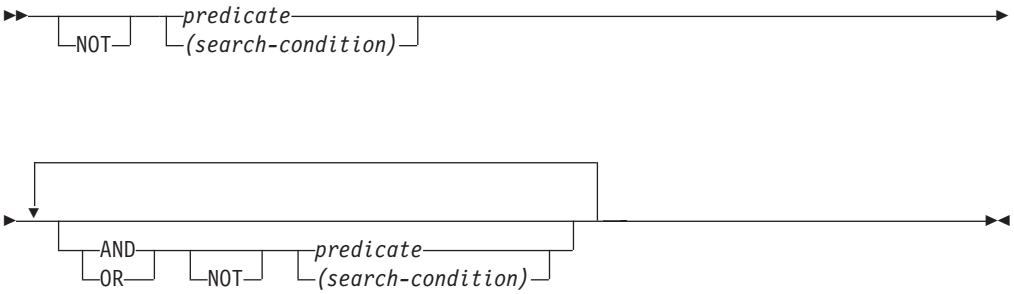
The result of a NULL predicate cannot be unknown. If the value of the column is null, the result is true. If the value is not null, the result is false.

If NOT is specified, the result is reversed.

Examples

```
EMPLOYEE.PHONE IS NULL

SALARY IS NOT NULL
```

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 19. 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

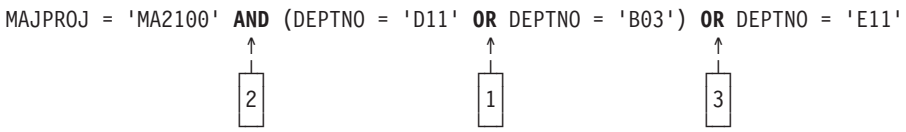
NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

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.

```
MAJPROJ = 'MA2100' AND DEPTNO = 'D11' OR DEPTNO = 'B03' OR DEPTNO = 'E11'
                   ↑                  ↑                 ↑
                   1                2 or 3            2 or 3
```


MAJPROJ = 'MA2100' AND (DEPTNO = 'D11' OR DEPTNO = 'B03') OR DEPTNO = 'E11'
2
1
3

Chapter 4. Built-in global variables

This chapter contains semantic descriptions, rules, and examples of the use of the *built-in global variables*.

Built-in global variables are provided with the database manager and are used in SQL statements to retrieve scalar values associated with the variables.

As an example, the PACKAGE_NAME global variable can be referenced in an SQL statement to retrieve the current package name.

The authorization ID of any statement that retrieves the value of the global variable is required to have the READ privilege on the global variable.

Example

To access the global variable CLIENT_IPADDR, run the following query:

```
SELECT SYSIBM.CLIENT_IPADDR
FROM SYSIBM.SYSDUMMY1
```

The query returns the TCP/IP address of the current client.

```
9.10.112.35
```

CLIENT_IPADDR

This global variable contains the IP address of the current client, as returned by the system.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).

  G In DB2 for z/OS, the type is CHAR(39).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the client did not connect by using the TCP/IP or SSL protocol, the value of the global variable is NULL.

PACKAGE_NAME

This global variable contains the name of the currently executing package.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

If there is no package currently executing, the value is NULL.

In a nested execution scenario, where one package invokes another, the PACKAGE_NAME global variable contains the name of the immediate package context. For example, if package A checks the value of the PACKAGE_NAME variable, the value is A. If package A invokes package B and package B checks the value of the PACKAGE_NAME variable, the value is B.

In DB2 for z/OS, the value of PACKAGE_NAME can only be obtained by invoking the GETVARIABLE function.

PACKAGE_SCHEMA

This global variable contains the schema name of the currently executing package.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

If there is no package currently executing, the value is NULL.

In a nested execution scenario, where one package invokes another, the PACKAGE_SCHEMA global variable contains the schema name of the immediate package context. For example, if package X.A checks the value of the PACKAGE_SCHEMA variable, the schema value is X. If package X.A invokes package Y.B and package B checks the value of the PACKAGE_SCHEMA variable, the schema value is Y.

G In DB2 for z/OS, the value of PACKAGE_SCHEMA can only be obtained by
G invoking the GETVARIABLE function.

PACKAGE_VERSION

This global variable contains the version identifier of the currently executing package.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(64).
- The schema is SYSIBM.
- The scope of this global variable is session.

If there is no package currently executing or the current executing package does not have a version identifier, the value is NULL.

In a nested execution scenario, where one package invokes another, the PACKAGE_VERSION global variable contains the version identifier of the immediate package context. For example, if package A checks the value of the PACKAGE_VERSION variable, then the value is 1.0. If package A invokes package B and package B checks the value of the PACKAGE_VERSION variable, then the value is 1.8.

G In DB2 for z/OS, the value of PACKAGE_VERSION can only be obtained by
G invoking the GETVARIABLE function.

Chapter 5. 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 on functions, see "Functions" on page 123.

Table 20. Aggregate Functions

Function	Description	Reference
ARRAY_AGG	Aggregates a set of elements into an array	"ARRAY_AGG" on page 196
AVG	Returns the average of a set of numbers	"AVG" on page 198
COUNT	Returns the number of rows or values in a set of rows or values	"COUNT" on page 200
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 201
GROUPING	Returns a value that indicates whether a row was generated as a summary row for a grouping set.	"GROUPING" on page 202
MAX	Returns the maximum value in a set of values in a group	"MAX" on page 204
MIN	Returns the minimum value in a set of values in a group	"MIN" on page 205
STDDEV	Returns the biased standard deviation of a set of numbers	"STDDEV" on page 206
SUM	Returns the sum of a set of numbers	"SUM" on page 207
VARIANCE or VAR	Returns the biased variance of a set of numbers	"VARIANCE or VAR" on page 208
XMLAGG	Returns an XML sequence containing an item for each non-null value in a set of XML values	"XMLAGG" on page 209

Table 21. Cast Scalar Functions

Function	Description	Reference
BIGINT	Returns a big integer representation of a number	"BIGINT" on page 221
BLOB	Returns a BLOB representation of a string of any type	"BLOB" on page 224
CHAR	Returns a CHARACTER representation of a value	"CHAR" on page 227
CLOB	Returns a CLOB representation of a value	"CLOB" on page 234
DATE	Returns a DATE from a value	"DATE" on page 245
DBCLOB	Returns a DBCLOB representation of a string	"DBCLOB" on page 253
DECFLOAT	Returns a DECFLOAT representation of a number	"DECFLOAT" on page 255
DECIMAL	Returns a DECIMAL representation of a number	"DECIMAL or DEC" on page 257
DOUBLE_PRECISION or DOUBLE	Returns a DOUBLE PRECISION representation of a number	"DOUBLE_PRECISION or DOUBLE" on page 265
FLOAT	Returns a FLOAT representation of a number	"FLOAT" on page 273

Table 21. Cast Scalar Functions (continued)

Function	Description	Reference
GRAPHIC	Returns a GRAPHIC representation of a string	"GRAPHIC" on page 277
INTEGER or INT	Returns an INTEGER representation of a number	"INTEGER or INT" on page 289
REAL	Returns a REAL representation of a number	"REAL" on page 344
SMALLINT	Returns a SMALLINT representation of a number	"SMALLINT" on page 369
TIME	Returns a TIME from a value	"TIME" on page 382
TIMESTAMP	Returns a TIMESTAMP from a value or a pair of values	"TIMESTAMP" on page 383
TIMESTAMP_ISO	Returns a timestamp value from a datetime value	"TIMESTAMP_ISO" on page 388
VARCHAR	Returns a VARCHAR representative of a value	"VARCHAR" on page 404
VARGRAPHIC	Returns a VARGRAPHIC representation of a value	"VARGRAPHIC" on page 412

Table 22. 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 214
DAY	Returns the day part of a value	"DAY" on page 247
DAYNAME	Returns the name of the day part of a value	"DAYNAME" on page 248
DAYOFWEEK	Returns the day of the week from a value, where 1 is Sunday and 7 is Saturday	"DAYOFWEEK" on page 249
DAYOFWEEK_ISO	Returns the day of the week from a value, where 1 is Monday and 7 is Sunday	"DAYOFWEEK_ISO" on page 250
DAYOFYEAR	Returns the day of the year from a value	"DAYOFYEAR" on page 251
DAYS	Returns an integer representation of a date	"DAYS" on page 252
EXTRACT	Returns a specified portion of a datetime value	"EXTRACT" on page 271
HOUR	Returns the hour part of a value	"HOUR" on page 281
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 291
LAST_DAY	Returns a date or timestamp that represents the last day of the month of the argument	"LAST_DAY" on page 292
MICROSECOND	Returns the microsecond part of a value	"MICROSECOND" on page 307
MIDNIGHT_SECONDS	Returns an integer value representing the number of seconds between midnight and a specified time value	"MIDNIGHT_SECONDS" on page 308
MINUTE	Returns the minute part of a value	"MINUTE" on page 310
MONTH	Returns the month part of a value	"MONTH" on page 313
MONTHNAME	Returns the name of the month part of a value	"MONTHNAME" on page 314
MONTHS_BETWEEN	Returns an estimate of the number of months between two dates	"MONTHS_BETWEEN" on page 315

Table 22. Datetime Scalar Functions (continued)

Function	Description	Reference
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 329
QUARTER	Returns an integer that represents the quarter of the year in which a date resides	"QUARTER" on page 340
ROUND_TIMESTAMP	Returns a timestamp rounded to the specified unit	"ROUND_TIMESTAMP" on page 354
SECOND	Returns the seconds part of a value	"SECOND" on page 364
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 385
TIMESTAMPDIFF	Returns an estimated number of intervals based on the difference between two timestamps	"TIMESTAMPDIFF" on page 389
TRUNC_TIMESTAMP	Returns a timestamp truncated to the specified unit	"TRUNC_TIMESTAMP" on page 400
VARCHAR_FORMAT	Returns a character string representation of a timestamp, with the string in a specified format	"VARCHAR_FORMAT" on page 409
WEEK	Returns the week of the year from a value, where the week starts with Sunday	"WEEK" on page 416
WEEK_ISO	Returns the week of the year from a value, where the week starts with Monday	"WEEK_ISO" on page 417
YEAR	Returns the year part of a value	"YEAR" on page 418

Table 23. Miscellaneous Scalar Functions

Function	Description	Reference
CARDINALITY	Returns a value representing the number of elements of an array	"CARDINALITY" on page 225
COALESCE	Returns the first argument that is not null	"COALESCE" on page 236
CONTAINS	Returns an indication of whether a match was found in a text index	"CONTAINS" on page 240
GENERATE_UNIQUE	Returns a bit character string that is unique compared to any other execution of the function	"GENERATE_UNIQUE" on page 275
HEX	Returns a hexadecimal representation of a value	"HEX" on page 280
IDENTITY_VAL_LOCAL	Returns the most recently assigned value for an identity column	"IDENTITY_VAL_LOCAL" on page 282
LENGTH	Returns the length of a value	"LENGTH" on page 295
MAX	Returns the maximum value in a set of values	"MAX" on page 305
MAX_CARDINALITY	Returns a value representing the maximum number of elements an array can contain	"MAX_CARDINALITY" on page 306
MIN	Returns the minimum value in a set of values	"MIN" on page 309
NULLIF	Returns a null value if the arguments are equal, otherwise it returns the value of the first argument	"NULLIF" on page 332
RAISE_ERROR	Raises an error with the specified SQLSTATE and message text	"RAISE_ERROR" on page 342
RID	Returns the RID of a row as BIGINT	"RID" on page 349

Table 23. Miscellaneous Scalar Functions (continued)

Function	Description	Reference
SCORE	Returns an indication of how frequently a match was found in a text index	"SCORE" on page 361
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 397
VALUE	Returns the first argument that is not null	"VALUE" on page 403
VERIFY_GROUP_FOR_USER	Returns an indication of whether a specified user is part of a group.	"VERIFY_GROUP_FOR_USER" on page 414

Table 24. 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 317
MQREADCLOB	Returns a message from a specified MQSeries location (return value of CLOB) without removing the message from the queue	"MQREADCLOB" on page 319
MQRECEIVE	Returns a message from a specified MQSeries location (return value of VARCHAR) with removal of message from the queue	"MQRECEIVE" on page 321
MQRECEIVECLOB	Returns a message from a specified MQSeries location (return value of CLOB) with removal of message from the queue	"MQRECEIVECLOB" on page 323
MQSEND	Sends a message to a specified MQSeries location	"MQSEND" on page 325

Table 25. Numeric Scalar Functions

Function	Description	Reference
ABS	Returns the absolute value of a number	"ABS" on page 212
ACOS	Returns the arc cosine of a number, in radians	"ACOS" on page 213
ASIN	Returns the arc sine of a number, in radians	"ASIN" on page 217
ATAN	Returns the arc tangent of a number, in radians	"ATAN" on page 218
ATANH	Returns the hyperbolic arc tangent of a number, in radians	"ATANH" on page 219
ATAN2	Returns the arc tangent of x and y coordinates as an angle expressed in radians	"ATAN2" on page 220
BITAND, BITANDNOT, BITOR, BITXOR, and 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 222
CEILING	Returns the smallest integer value that is greater than or equal to a number	"CEILING" on page 226
COMPARE_DECFLOAT	Returns an indication of how two decimal floating-point values compare	"COMPARE_DECFLOAT" on page 237
COS	Returns the cosine of a number	"COS" on page 243
COSH	Returns the hyperbolic cosine of a number	"COSH" on page 244
DEGREES	Returns the number of degrees of an angle	"DEGREES" on page 262

Table 25. Numeric Scalar Functions (continued)

Function	Description	Reference
DIGITS	Returns a character-string representation of the absolute value of a number	"DIGITS" on page 264
EXP	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument	"EXP" on page 270
FLOOR	Returns the largest integer value that is less than or equal to a number	"FLOOR" on page 274
LN	Returns the natural logarithm of a number	"LN" on page 296
LOG10	Returns the common logarithm (base 10) of a number	"LOG10" on page 299
MOD	Returns the remainder of the first argument divided by the second argument	"MOD" on page 311
MULTIPLY_ALT	Multiplies the first argument by the second argument and returns the product	"MULTIPLY_ALT" on page 327
NORMALIZE_DECFLOAT	Returns a decimal floating-point value in its simplest form	"NORMALIZE_DECFLOAT" on page 331
POWER	Returns the result of raising the first argument to the power of the second argument	"POWER" on page 337
QUANTIZE	Returns a decimal floating-point value formatted according to a provided value	"QUANTIZE" on page 338
RADIANS	Returns the number of radians for an argument that is expressed in degrees	"RADIANS" on page 341
RAND	Returns a random number	"RAND" on page 343
ROUND	Returns a numeric value that has been rounded to the specified number of decimal places	"ROUND" on page 352
SIGN	Returns the sign of a number	"SIGN" on page 366
SIN	Returns the sine of a number	"SIN" on page 367
SINH	Returns the hyperbolic sine of a number	"SINH" on page 368
SQRT	Returns the square root of a number	"SQRT" on page 372
TAN	Returns the tangent of a number	"ATAN" on page 218
TANH	Returns the hyperbolic tangent of a number	"TANH" on page 381
TOTALORDER	Returns an ordering indication for two decimal floating-point values	"TOTALORDER" on page 392
TRUNCATE or TRUNC	Returns a number value that has been truncated at a specified number of decimal places	"TRUNCATE or TRUNC" on page 398

Table 26. 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 216
CHARACTER_LENGTH	Returns the length of a string expression	"CHARACTER_LENGTH" on page 233
CONCAT	Returns a string that is the concatenation of two strings	"CONCAT" on page 239

Table 26. String Scalar Functions (continued)

Function	Description	Reference
DECRYPT_BIT and DECRYPT_CHAR	Decrypts an encrypted string	"DECRYPT_BIT and DECRYPT_CHAR" on page 260
DIFFERENCE	Returns a value representing the difference between the sounds of two strings	"DIFFERENCE" on page 263
ENCRYPT	Encrypts a string	"ENCRYPT" on page 267
GETHINT	Returns a hint from an encrypted string	"GETHINT" on page 276
INSERT	Returns a string where a substring is deleted and a new string inserted in its place	"INSERT" on page 287
LCASE	Returns a string in which all the characters have been converted to lowercase characters	"LCASE" on page 293
LEFT	Returns the leftmost characters from the string	"LEFT" on page 294
LOCATE	Returns the starting position of one string within another string	"LOCATE" on page 297
LOWER	Returns a string in which all the characters have been converted to lowercase characters	"LOWER" on page 300
LPAD	Returns a string composed of *expression* that is padded on the left	"LPAD" on page 301
LTRIM	Returns a string in which blanks have been removed from the beginning of another string	"LTRIM" on page 304
POSITION	Returns the starting position of one string within another string	"POSITION" on page 333
POSSTR	Returns the starting position of one string within another string	"POSSTR" on page 335
REPEAT	Returns a string composed of another string repeated a number of times	"REPEAT" on page 345
REPLACE	Returns a string where all occurrences of one string are replaced by another string	"REPLACE" on page 347
RIGHT	Returns the rightmost characters from the string	"RIGHT" on page 350
RPAD	Returns a string composed of *expression* that is padded on the right	"RPAD" on page 357
RTRIM	Returns a string in which blanks have been removed from the end of another string	"RTRIM" on page 360
SOUNDEX	Returns a character code representing the sound of the words in the argument	"SOUNDEX" on page 370
SPACE	Returns a character string that consists of a specified number of blanks	"SPACE" on page 371
STRIP	Removes blanks or another specified character from the end or beginning of a string expression	"STRIP" on page 373
SUBSTR	Returns a substring of a string	"SUBSTR" on page 375
SUBSTRING	Returns a substring of a string	"SUBSTRING" on page 378
TRANSLATE	Returns a string in which one or more characters in a string are converted to other characters	"TRANSLATE" on page 393
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 395

Table 26. String Scalar Functions (continued)

Function	Description	Reference
UCASE	Returns a string in which all the characters have been converted to uppercase characters	"UCASE" on page 401
UPPER	Returns a string in which all the characters have been converted to uppercase characters	"UPPER" on page 402

Table 27. XML Scalar Functions

Function	Description	Reference
XMLATTRIBUTES	Constructs XML attributes from the arguments.	"XMLATTRIBUTES" on page 419
XMLCOMMENT	Returns an XML value with the input argument as the content.	"XMLCOMMENT" on page 420
XMLCONCAT	Returns an XML sequence containing the concatenation of a variable number of XML input arguments.	"XMLCONCAT" on page 421
XMLDOCUMENT	Returns an XML value that is a well-formed XML document.	"XMLDOCUMENT" on page 423
XMLELEMENT	Returns an XML value that is an XML element.	"XMLELEMENT" on page 424
XMLFOREST	Returns an XML value that is a sequence of XML elements.	"XMLFOREST" on page 428
XMLNAMESPACES	Constructs namespace declarations from the arguments.	"XMLNAMESPACES" on page 431
XMLPARSE	Returns an XML value by parsing the arguments as an XML document.	"XMLPARSE" on page 433
XMLPI	Returns an XML value with a single processing instruction.	"XMLPI" on page 435
XMLSERIALIZE	Returns a serialized XML value as the specified data type.	"XMLSERIALIZE" on page 436
XMLTEXT	Returns an XML value that contains the value of the input argument.	"XMLTEXT" on page 438
XSLTRANSFORM	Transforms an XML document into a different data format.	"XSLTRANSFORM" on page 439

Table 28. Table Functions

Function	Description	Reference
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 443
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 446
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 449

Table 28. Table Functions (continued)

Function	Description	Reference
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 452
XMLTABLE	Returns a table from the evaluation of an XPath expression.	"XMLTABLE" on page 455

Aggregate functions

The following information applies to all aggregate functions. However, it does not apply when an asterisk (*) is used as the argument to COUNT or COUNT_BIG.

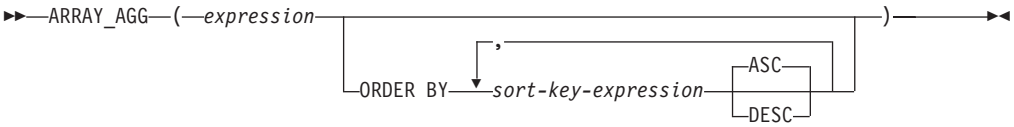
- The argument of an aggregate function is a set of values derived from an expression. The expression must not include another aggregate function or a *scalar-fullselect*. The scope of the set is a group or an intermediate result table as explained in Chapter 6, "Queries," on page 463.
- If a GROUP BY clause is specified in a query and the intermediate result of the FROM, WHERE, GROUP BY, and HAVING clauses is an empty result table; then the aggregate functions are not applied, the result of the query is an empty table.
- If a GROUP BY clause is not specified in a query and the intermediate result of the FROM, WHERE, and HAVING clauses is an empty result table, then the aggregate functions are applied to the empty table. For example, the result of the following SELECT statement is applied to the empty result table 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 an 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 16 000. See Table 63 on page 967 for more information.

The ARRAY_AGG function can only be specified within an SQL procedure or an SQL function in the following specific contexts:

- The *select-clause* of a SELECT INTO statement
- The *select-clause* of a scalar subquery on the right side of a SET statement
- The *select-clause* of the outermost fullselect in the definition of a cursor that is not scrollable
- A RETURN statement in an SQL scalar function

The following restrictions apply to ARRAY_AGG:

- A fullselect that contains an invocation of ARRAY_AGG cannot contain a DISTINCT keyword in its *select-clause*.
- ARRAY_AGG cannot be used as part of an OLAP specification.
- A fullselect that contains an invocation of ARRAY_AGG cannot contain an ORDER BY clause.
- The SELECT clause or HAVING clause of the fullselect that contains an invocation of ARRAY_AGG cannot contain a subquery.
- A SELECT clause that includes an invocation of the ARRAY_AGG function that returns an array of LOBs must not also include a GROUP BY clause.

Example

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) )
```

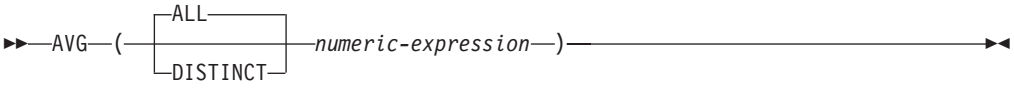
Create a procedure that uses a SELECT INTO statement to return the prioritized list of contact numbers under which an employee can be reached.

```
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
An expression that returns a value of 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:

- G The result is DECFLOAT(34) if the argument values are DECFLOAT(16). In DB2 G for LUW the result is DECFLOAT((16).
- The result is double-precision floating point if the argument values are single-precision floating point.
- The result is a large integer if the argument values are small integers.
- The result is decimal with precision 31 and scale 31-*p*+*s* if the argument values G are decimal numbers with precision *p* and scale *s*.[36] In DB2 for z/OS, the scale is G MAX(0,28-*p*+*s*).

The function is applied to the set of values derived from the argument values by eliminating null values. If DISTINCT is specified, redundant duplicate values are eliminated.

The result can be null. If the 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.

Note

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).

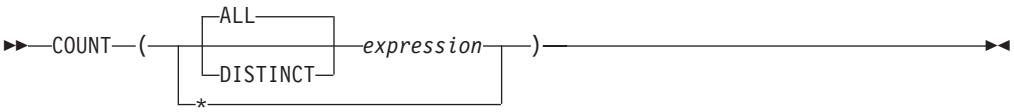
36. For DB2 for z/OS, the formulas used in this book are those that apply when the DEC31 option is in effect or the precision of an operand is greater than 15.

- Using the PROJECT table, set the host variable ANY_CALC to the average of each unique staffing level 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
An expression that returns a value of any built-in data type other than a BLOB, CLOB, DBCLOB, or XML.

The result of the function is a large integer and must be within the range of large integers. The result cannot be null.

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 of values 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 redundant duplicate values. The result is the number of different non-null values in the set.

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'
```

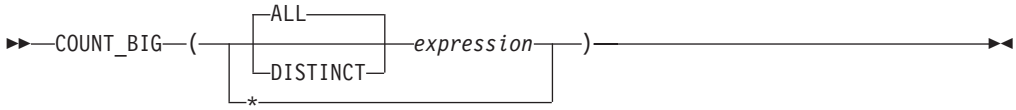
  Results in FEMALE being set to 19.

- 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
An expression that returns a value of any built-in data type other than a BLOB, CLOB, DBCLOB, or XML.

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*) or COUNT_BIG(ALL *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. The result is the number of non-null values in the set including duplicates.

The argument of COUNT_BIG(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 redundant duplicate values. The result is the number of different non-null values in the set.

Examples

- Refer to the COUNT examples and substitute COUNT_BIG for occurrences of COUNT. The results are the same except for the data type of the result.
- To create a sourced function that is similar to the built-in COUNT_BIG function, the definition of the sourced function must include the type of the column that can be specified when the new function is invoked. 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 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 previously listed value.

Example

The following query:

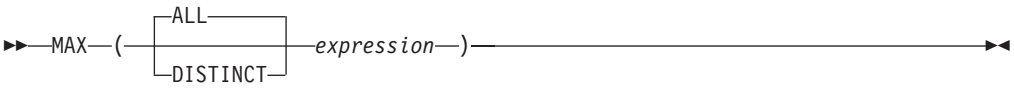
```
SELECT SALES_DATE, SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD,
       GROUPING(SALES_DATE) AS DATE_GROUP,
       GROUPING(SALES_PERSON) AS SALES_GROUP
FROM SALES
GROUP BY CUBE( SALES_DATE, SALES_PERSON)
ORDER BY SALES_DATE, SALES_PERSON
```

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 function returns the maximum value in a set of values.

expression
An expression that returns a value of any built-in data type other than a BLOB, CLOB, DBCLOB, 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 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 set of values is empty, the result is the 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.

Note

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 86. 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
```

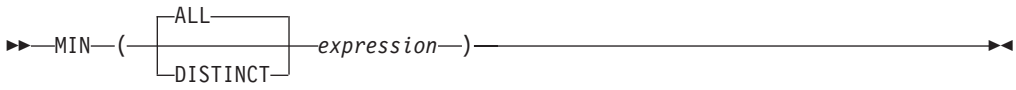
  Results in MAX_SALARY being set to 4395.83.

- Using the PROJECT table, set the host variable LAST_PROJ (CHAR(24)) to the project name (PROJNAME) that comes last in the collating sequence.

```
SELECT MAX(PROJNAME)
  INTO :LAST_PROJ
  FROM PROJECT
```

  Results in LAST_PROJ being set to 'WELD LINE PLANNING'.

MIN



The MIN function returns the minimum value in a set of values.

expression
An expression that returns a value of any built-in data type other than a BLOB, CLOB, DBCLOB, 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 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 set of values is empty, the result is the 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.

Note

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 86. 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

The STDDEV function returns the biased population standard deviation (/n) of a set of numbers. The formula used to calculate STDDEV is logically equivalent to:

```
STDDEV = SQRT(VAR)
```

where SQRT(VAR) is the square root of the variance.

numeric-expression
An expression that returns a value of any built-in numeric data type.

If the argument is DECFLOAT(n), the result of the function is DECFLOAT(34).
G Otherwise, the data type of the result is double-precision floating point. In DB2 for
G LUW if the argument is DECFLOAT(16), the result is DECFLOAT((16).

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated.

The result can be null. If the set of values is empty, 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 aggregated is undefined, but every intermediate result must be within the range of the result data type.

Note

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.

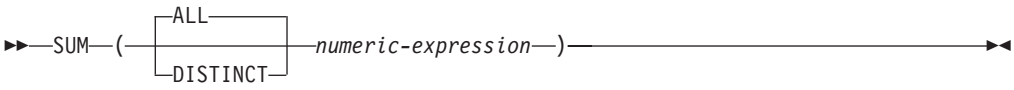
Example

- Using the EMPLOYEE table, set the host variable DEV (double-precision floating point) to the standard deviation of the salaries for those employees in department A00.

```
SELECT STDDEV(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';
```

  Results in DEV being set to approximately 9742.43.

SUM



The SUM function returns the sum of a set of numbers.

numeric-expression
An expression that returns a value of 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:

- G DECFLOAT(34) if the argument values are DECFLOAT(16). In DB2 for LUW the G result is 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 31 and scale *s* if the argument values are decimal G numbers with precision *p* and scale *s*.[37] In DB2 for z/OS, the precision of the G result is min(31, *p*+10).

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated.

The result can be null. If the set of values is empty, the result is the null value. Otherwise, the result is the sum of the values in 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.

Note

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

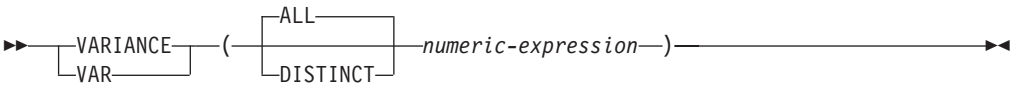
- Using the EMPLOYEE table, set the host variable JOB_BONUS (DECIMAL(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').

```
SELECT SUM(BONUS)
  INTO :JOB_BONUS
  FROM EMPLOYEE
  WHERE JOB = 'CLERK'
```

  Results in JOB_BONUS being set to 4000.

37. For DB2 for z/OS, the formulas used in this book are those that apply when the DEC31 option is in effect or the precision of an operand is greater than 15.

VARIANCE or VAR



The VARIANCE functions return the biased population variance (/n) of a set of numbers. The formula used to calculate VARIANCE is logically equivalent to:

```
VARIANCE = SUM(X**2)/COUNT(X) - (SUM(X)/COUNT(X))**2
```

numeric-expression
An expression that returns a value of 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. In DB2 for LUW if the argument is DECFLOAT(16), the result is DECFLOAT((16).

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated.

The result can be null. If the set of values is empty, 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 aggregated is undefined, but every intermediate result must be within the range of the result data type.

Note

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

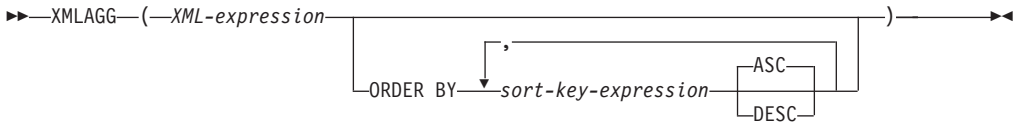
- Using the EMPLOYEE table, set the host variable VARNCE (double-precision floating point) to the variance of the salaries for those employees in department A00.

```
SELECT VARIANCE(SALARY)
  INTO :VARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';
```

  Results in VARNCE being set to approximately 94 915 000.

XMLAGG

The XMLAGG function returns an XML sequence containing an item for each non-null value in a set of XML values.



The function name cannot be specified as a qualified name.

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 LOB or an 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.

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 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.

Examples

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>
```

Scalar functions

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 set of 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 a aggregate function only if a 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.

numeric-expression
An expression that returns a value of any built-in numeric data type.

The data type and length attribute of the result are the same as the data type and length attribute of the argument value. In DB2 for i the result is an INTEGER if the argument value is a small integer and the result is double-precision floating point if the argument value is single-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.

Notes

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.

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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. 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

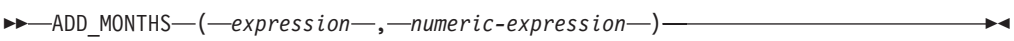
- Assume the host variable ACOSINE is a DECIMAL(10,9) host variable with a value of 0.070737202.

```
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.[38]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

numeric-expression
An expression that returns a value of a built-in numeric data type with zero scale. A negative numeric value is allowed.

If *expression* is a timestamp, the result of the function is a timestamp with the same precision as expression. 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*.

Examples

- 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)
  ```

38. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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.

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 string data type, except for CLOB and DBCLOB.[39] For a VARCHAR, the maximum length is 4 000 bytes. The first character of the string will be converted to an ASCII CCSID for processing by the function, if necessary.

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

- Return the integer value for the ASCII representation of 'A'.

```
SELECT ASCII('A')
   FROM SYSIBM.SYSDUMMY1
```

  Returns the value 65.

39. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. 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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. 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.

numeric-expression-1
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value must not be 0.

numeric-expression-2
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value 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 1.107 148 7.

BIGINT

Numeric to Big Integer

String to Big Integer

The BIGINT function returns a big integer representation of:

- A number
- A character-string representation of an integer
- A graphic-string representation of an integer

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 or graphic-string representation of an integer. The expression must not be a CLOB or DBCLOB and must have a length attribute that is not greater than 255 bytes.[40]

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 an integer constant. If the whole part of the argument is not within the range of big integers, an error is returned.

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 for maximal portability. For more information, see "CAST specification" on page 149.

Example

- Using the EMPLOYEE table, select the SALARY column in big integer form for further processing in the application.

```
SELECT BIGINT(SALARY)
  FROM EMPLOYEE
```

40. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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 returns the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.

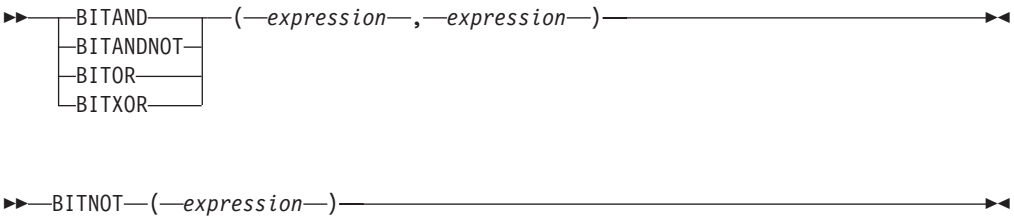


Table 29. Bit manipulation functions

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)).

Examples

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.

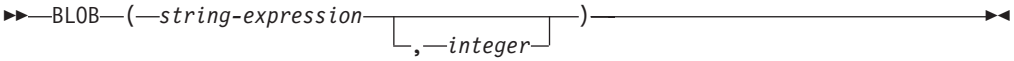
```
UPDATE ITEM
 SET PROPERTIES = BITXOR(PROPERTIES, 1024)
 WHERE ITEMID = 3412
```

- 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).

BLOB



The BLOB function returns a BLOB representation of a string of any type.

string-expression
An expression that returns a value that is a built-in character, graphic, or binary string data type.

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. For more information, see Table 60 on page 964.

If *integer* is not specified, 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. If *integer* is not specified, the *string-expression* must not be the empty string constant.

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.

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: When the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

Examples

- 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 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 array index 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 array type PHONENUMBERS and array variable RECENT_CALLS are defined as follows:

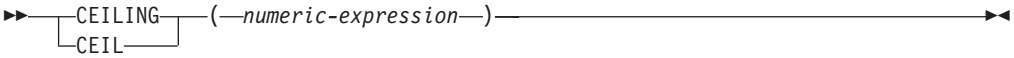
```
CREATE TYPE PHONENUMBERS AS INTEGER ARRAY[50];
DECLARE RECENT_CALLS PHONENUMBERS;
```

RECENT_CALLS contains three elements. The following SET statement assigns the number of calls that have been stored in the array so far to SQL variable HOWMANYCALLS:

```
SET HOWMANYCALLS = CARDINALITY(RECENT_CALLS)
```

After the statement executes, HOWMANYCALLS contains 3.

CEILING



The CEIL or CEILING function returns the smallest integer value that is greater than or equal to *numeric-expression*.

numeric-expression
An expression that returns a value of any built-in numeric data type.

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 DECIMAL. 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.

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 000004396. 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
```

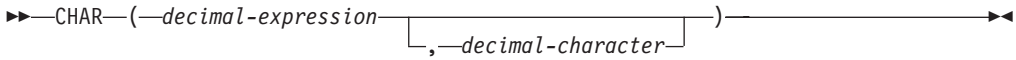
  This example returns (leading zeroes are shown to demonstrate the precision and scale of the result):

```
04.   04.  -03.  -03.
```

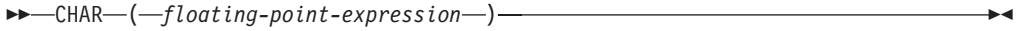
CHAR

Integer to Character

Decimal to Character

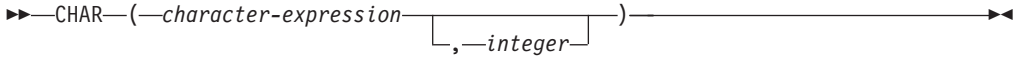


Floating-point to Character

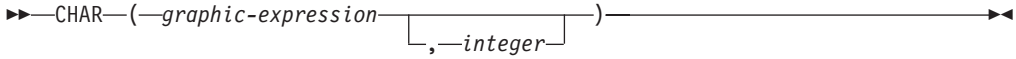


Decimal floating-point to Character

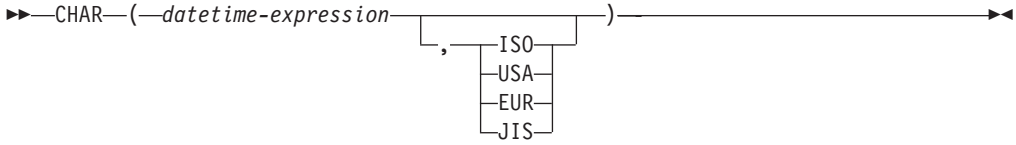
Character to Character



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.

The first argument must be a built-in data type other than a BLOB.

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 SMALLINT, INTEGER, or BIGINT data type.

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 or NUMERIC data type. If a different precision and scale is desired, 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 100.

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. If the scale of *decimal-expression* is zero, the decimal character is not 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 at least 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).

The single-byte character constant used to delimit the decimal digits in *character-expression* from the whole part of the number is the default decimal point. For more information, see "Decimal point" on page 100.

The result is a fixed-length character-string representation of the argument in the form of a floating-point constant. The length attribute 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 default decimal point. 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 period 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).

The single-byte character constant used to delimit the decimal digits in *character-expression* from the whole part of the number is the default decimal point. For more information, see "Decimal point" on page 100.

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 default decimal point. 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 default decimal point. 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.

integer
An integer constant that specifies the length attribute for the resulting fixed length character string. The value must be between 1 and 254. In EBCDIC environments, if the first argument is mixed data, the second argument cannot be less than 4.

If *integer* is not specified, the length of the result is the minimum of 254 and the length attribute of *character-expression*. The *character-expression* must not be the empty string constant.

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.

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.[41]

integer
An integer constant that specifies the length attribute for the resulting fixed length character string. The value must be between 1 and 254.

If *integer* is not specified, the length of the result is the minimum of 254 and the length attribute of *graphic-expression*. The *graphic-expression* must not be the empty string constant.

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.

The CCSID of the string is the default CCSID at 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 specified, the format used is the default date format. The length of the result is 10. For more information see "String representations of datetime values" on page 66.

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 from the default time format. The length of the result is 8. For more information see "String representations of datetime values" on page 66.

TIMESTAMP
The second argument is not applicable and must not be specified.

41. In DB2 for LUW, a *graphic-expression* is only allowed in a Unicode database.

The result is the character-string representation of the timestamp. If *datetime-expression* is a TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is a TIMESTAMP(*n*), where *n* is greater than 0, the length of the result is 20+*n*. Otherwise 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 66.

The CCSID of the string is the default SBCS CCSID at the current server.

Note

Syntax alternatives: When the first argument is numeric, or the first argument is a string and the length attribute is specified; the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

Examples

- Assume the column PRSTDATE has an internal value equivalent to 1988-12-25.

```
SELECT CHAR(PRSTDATE, USA)
   FROM PROJECT
```

  Results in the value '12/25/1988'.

- 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 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 '18bbbb' (blank padded on the right with 4 blanks).

- Assume the same SALARY column 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).

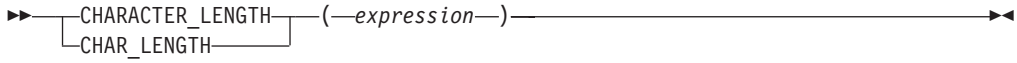
- Assume a host variable, DOUBLE_NUM, has a double-precision floating-point data type and a value of -987.654321E-35.

```
SELECT CHAR(:DOUBLE_NUM)
   FROM SYSIBM.SYSDUMMY1
```

Results in the character value '-9.8765432100000002E-33 '.[42]

42. Note that since floating-point numbers are approximate, the resulting character string will vary slightly based on that approximation.

CHARACTER_LENGTH



The CHARACTER_LENGTH or CHAR_LENGTH function returns the length of a string expression. See “LENGTH” on page 295 for a similar function.

expression
An expression that returns a value of any built-in character or graphic string data type. The *expression* cannot be bit data.

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 number of characters in the argument (not the number of bytes). A single character is either an SBCS, DBCS, or multiple-byte character. The length of strings includes trailing blanks. The length of a varying-length string is the actual length, not the maximum length.

G In DB2 for z/OS and DB2 for LUW a second argument is required. CODEUNITS32
G should be specified for the second argument.

Example

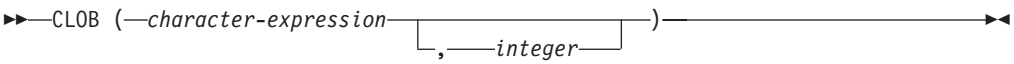
- Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'.

```
SELECT CHARACTER_LENGTH(NAME), LENGTH(NAME)
  FROM T1
  WHERE NAME = 'Jürgen'
```

  Returns the value 6 for CHARACTER_LENGTH and 7 for LENGTH.

CLOB

Character to Character



Graphic to Character

The CLOB function returns a CLOB representation of:

- 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 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.

Character to Character

character-expression
An expression that returns a value that is a built-in character-string data type. The argument must not be bit data.

integer
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. In EBCDIC environments, if the first argument is mixed data, the second argument cannot be less than 4.

If *integer* is not specified the length attribute of the result is the same as the length attribute of the first argument. The *character-expression* must not be the empty string constant.

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.

The CCSID of the result is the same as the CCSID of the first argument.

Graphic to Character

graphic-expression
An expression that returns a value that is a built-in graphic-string data type.[43]

integer
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 *integer* is not specified the length attribute of the result is the same as the length attribute of the first argument. The *graphic-expression* must not be the empty string constant.

43. In DB2 for LUW, a *graphic-expression* is only allowed in a Unicode database.

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.

The CCSID of the string is the default CCSID at the current server.

Note

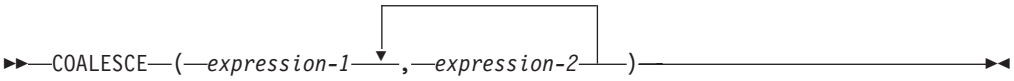
Syntax alternatives: When the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

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.

expression-1
An expression that returns a value of any built-in or user-defined data type.[44]

expression-2
An expression that returns a value of any built-in or user-defined data type.[44]

The arguments must be compatible. For more information on data type compatibility, see "Assignments and comparisons" on page 77.

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 91.

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'.

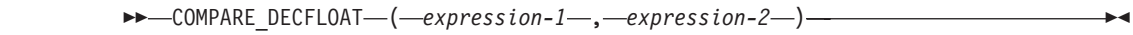
```
SELECT DEPTNO, DEPTNAME, COALESCE(MGRNO, 'ABSENT'), ADMRDEPT
  FROM DEPARTMENT
```

- 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
```

44. 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 specifies a DECFLOAT value. *expression-1* must be a built-in DECFLOAT value.

expression-2
An expression that specifies a DECFLOAT value. *expression-2* must be a built-in DECFLOAT value.

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 the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

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 string arguments.

string-expression-1
An expression that returns a value of any built-in character string, graphic string, or binary string data type.

string-expression-2
An expression that returns a value of any built-in character string, graphic string, or binary string data type.

The arguments must be compatible strings. For more information on data type compatibility, see "Assignments and comparisons" on page 77.

The result of the function is a string that consists of the first argument string followed by the second. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Notes

The CONCAT function is identical to the CONCAT operator. For more information, see "With the concatenation operator" on page 134.

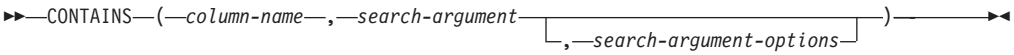
Example

- Concatenate the column FIRSTNME with the column LASTNAME.

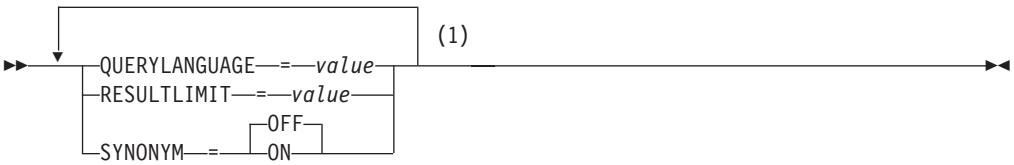
```
SELECT CONCAT(FIRSTNME, LASTNAME)
  FROM EMPLOYEE
  WHERE EMPNO = '000010'
```

  Returns the value 'CHRISTINEHAAS'.

CONTAINS



search-argument-options



Notes:

1 The same clause must not be specified more than once.

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.

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 is not a CLOB or DBCLOB 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 4 096 Unicode characters. The value is converted to Unicode before it is used to search the text search index. The value 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 the product documentation.

search-argument-options
A character string or graphic string constant that specifies the search argument options to use for the search. The options that can be specified as part of the *search-argument-options* are:

QUERYLANGUAGE = *value*
Specifies the language value. The value must be one 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 the product documentation.

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 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 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 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.

Note

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.

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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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.

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.[45]

- 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.
  - 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 greater than or equal to one and 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 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 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.

Note

Syntax alternatives: When the argument is a date, timestamp, or string, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

45. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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.[46]

- 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.
- 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 137.

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 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.

46. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

DAYNAME

The DAYNAME function returns a mixed case character string containing the name of the day (e.g. Friday) for the day portion of the argument.[47] The name of the day depends on the national language (locale). In DB2 for z/OS the name of the day is returned in English.

expression
An expression that returns a value of one of the following built-in data types: a date or a character string.

If *expression* is a character string, it must be a VARCHAR and its value must be a valid string representation of a date in ISO format with an actual length not larger than 10. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

The result of the function is VARCHAR(100). In DB2 for z/OS, the result of the function is VARCHAR(9). 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.

Examples

- Assume that the language used is US English.

```
SELECT DAYNAME('2003-01-02')
  FROM SYSIBM.SYSDUMMY1
```

Results in 'Thursday'.

47. In DB2 for z/OS, installation job DSNTEJ2U must be run to use this function.

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 250.

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.[48]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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(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(20))),
  FROM SYSIBM.SYSDUMMY1
```

48. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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" on page 249.

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.[49]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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(20))),
  FROM SYSIBM.SYSDUMMY1
```

49. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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.[50]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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.

50. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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.[51]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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'.

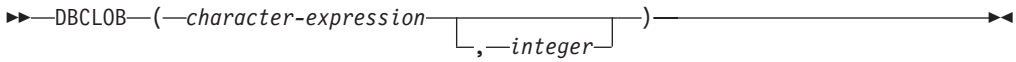
```
SELECT SUM(DAYS(PRENDATE) - DAYS(PRSTDATE))
  INTO :TOTAL_DAYS
  FROM PROJECT
  WHERE DEPTNO = 'E21'
```

  Results in TOTAL_DAYS being set to 1584.

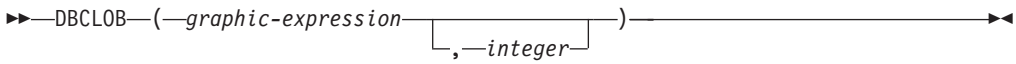
51. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

DBCLOB

Character to DBCLOB



Graphic to DBCLOB



The DBCLOB function returns a DBCLOB representation of:

- 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 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.

Character to DBCLOB

character-expression
An expression that returns a value that is a built-in character-string data type.[52]

integer
An integer constant that specifies the length attribute for the resulting varying-length graphic string. The value must be between 1 and 1 073 741 823.

If *integer* is not specified the length attribute of the result is the same as the length attribute of the first argument. The *character-expression* must not be the empty string constant.

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.

The CCSID of the result is determined by a mixed data CCSID. For more information, see Determining the Graphic Result CCSID.

Graphic to DBCLOB

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 varying-length graphic string. The value must be between 1 and 1 073 741 823.

If *integer* is not specified the length attribute of the result is the same as the length attribute of the first argument. The *graphic-expression* must not be the empty string constant.

52. In DB2 for LUW, a character string is only allowed in a Unicode database. If a supplied argument is a character string, it is first converted to a graphic string before the function is executed.

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.

The CCSID of the result is the same as the CCSID of the first argument.

Note

Syntax alternatives: When the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

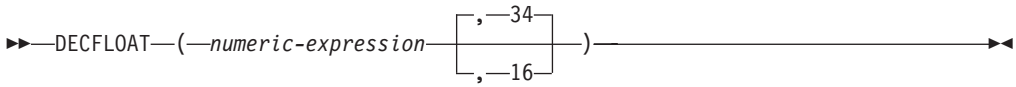
Example

- 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'.

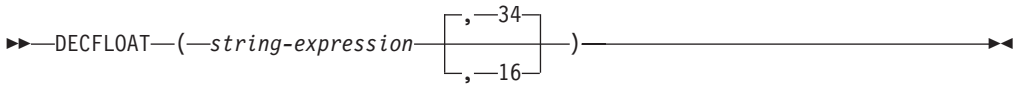
```
SELECT DBCLOB(VARGRAPHIC(FIRSTNME))
  INTO :VAR_DESC
  FROM EMPLOYEE
  WHERE EMPNO = '000050'
```

DECFLOAT

Numeric to DECFLOAT



String to DECFLOAT



The DECFLOAT function returns a decimal floating-point representation of:

- A number
- A character-string representation of a decimal number
- A character-string representation of an integer
- A character-string representation of a floating-point number
- A character-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 representation of a number. Leading and trailing blanks are eliminated and the resulting string is folded to uppercase and must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant. The expression must not be a CLOB and must have a length attribute that is not greater than 255 bytes.

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 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 103 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 149.

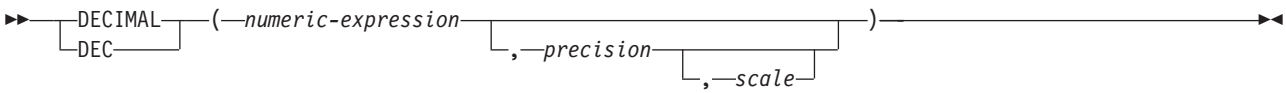
Examples

- 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-clause*.

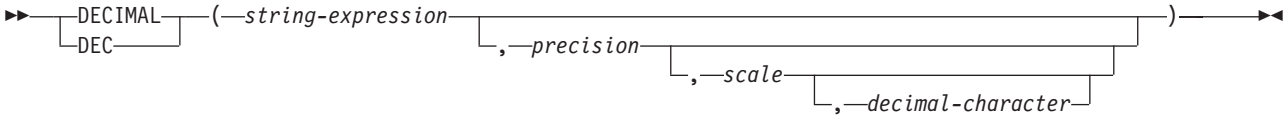
```
SELECT EMPNO, DECFLOAT(EDLEVEL,16)
  FROM EMPLOYEE
```

DECIMAL or DEC

Numeric to Decimal



String to Decimal



The DECIMAL function returns a decimal representation of:

- A number
- A character or graphic-string representation of an integer
- A character or graphic-string representation of a decimal 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 31.

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 or decimal
- 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*.

String to Decimal

string-expression
An expression that returns a string representation of a number. Leading and trailing blanks are eliminated and the resulting string must conform to the

rules for forming an integer or decimal constant. The expression must not be a CLOB or DBCLOB and must have a length attribute that is not greater than 255 bytes.[53]

precision
An integer constant that is greater than or equal to 1 and less than or equal to 31. 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 the *decimal-character* is not specified, the decimal point is the default decimal separator character. For more information, see "Decimal point" on page 100.

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 the scale *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 a different *decimal-character* 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: When the precision is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

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
```

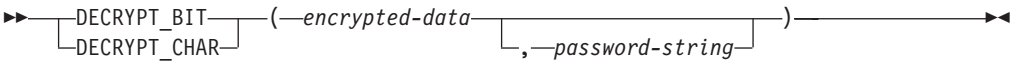
- Assume that updates to the SALARY column are input through a window as a character string using comma as a decimal character (for example, the user inputs 21400,50). Once validated by the application, it is assigned to the host variable newsalary which is defined as CHAR(10).

53. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

```
UPDATE STAFF
  SET SALARY = DECIMAL(:newsalary, 9, 2, ',')
  WHERE ID = :empid
```

The value of SALARY becomes 21 400.50.

DECRYPT_BIT and DECRYPT_CHAR



The DECRYPT_BIT and DECRYPT_CHAR 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.

G DB2 for LUW supports DECRYPT_BIN instead of DECRYPT_BIT.

encrypted-data
An expression that must be a string expression that returns a complete, encrypted data value of a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA built-in data type. The data string must have been encrypted using the ENCRYPT 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 an error is returned. 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.

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.

Function	Data Type of First Argument	Actual Data Type of Encrypted Data	Result
DECRYPT_BIT	CHAR FOR BIT DATA or VARCHAR FOR BIT DATA	Character string	VARCHAR FOR BIT DATA
DECRYPT_CHAR	CHAR FOR BIT DATA or VARCHAR FOR BIT DATA	Character string	VARCHAR

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 first argument can be null, the result can be null; if the first argument is null, the result is the null value.

If DECRYPT_BIT is specified, the CCSID of the result is 65535. If DECRYPT_CHAR is specified, the CCSID of the result is the default CCSID of the current server. 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 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.

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( '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('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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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.

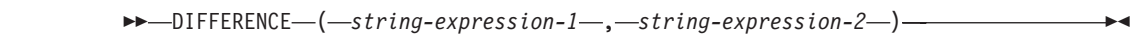
Example

- 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.

string-expression-1
An expression that returns a built-in character-string or graphic-string data type, but not a CLOB or DBCLOB. [54] The argument cannot be a binary string.

string-expression-2
An expression that returns a built-in character-string or graphic-string data type, but not a CLOB or DBCLOB. [54] The argument cannot be a binary string.

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.

54. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

DIGITS

The DIGITS function returns a character-string representation of the absolute value of a number.

numeric-expression
An expression that returns a value of one of the following built-in data types: SMALLINT, INTEGER, BIGINT, DECIMAL, or NUMERIC.

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
- 10 if the argument is a large integer
- 19 if the argument is a big integer
- *p* if the argument is a decimal number with a precision of *p*.

The CCSID of the character string is the default 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. Then, for this value:

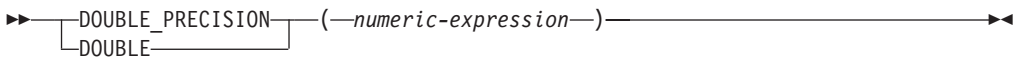
```
SELECT DIGITS(COLUMNX)
  FROM TABLEX
```

  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.

DOUBLE_PRECISION or DOUBLE

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 an integer
- A character or graphic-string representation of a decimal number
- A character or graphic-string representation of a 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 of a built-in character or graphic-string data type.[55] The argument must not be a CLOB or DBCLOB and must have a length attribute that is not greater than 255 bytes.

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, integer, or decimal constant.

The single-byte character constant used to delimit the decimal digits in *string-expression* from the whole part of the number must be the default decimal point. For more information, see "Decimal point" on page 100.

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.

Notes

Syntax alternatives: FLOAT can be specified in place of DOUBLE_PRECISION and DOUBLE.

The CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

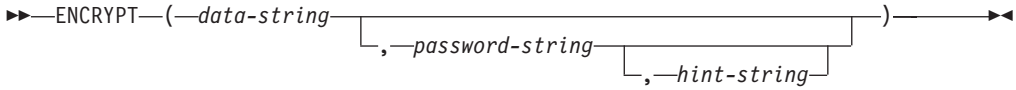
55. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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



The ENCRYPT function returns a value that is the result of encrypting *data-string*. The password used for encryption 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 character or graphic string value. The expression must not be a CLOB or DBCLOB.

G In DB2 for LUW, a graphic string is only allowed in a Unicode database. If a
G supplied argument is a graphic string, it is first converted to a character string
G before the function is executed.

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 this argument is the null value or is not provided, no hint will be embedded in the result.

The data type of the result is VARCHAR FOR BIT DATA.

The length attribute of the result is the length attribute of *data-string* plus *n*, where *n* is the amount of overhead necessary to encrypt the value.

- If a hint is not specified, *n* is 8 bytes + the number of bytes to the next 8 byte boundary (8 - MOD(LENGTH(*data-string*), 8)).
- If a hint is specified, *n* is 8 bytes + the number of bytes to the next 8 byte boundary (8 - MOD(LENGTH(*data-string*), 8)) + 32 bytes for the hint length.

The actual length of the result is the actual length of *data-string* plus *n*.

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 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.

G **Encryption algorithm:** The internal encryption algorithm used is product-specific.
G The length calculations above use RC2 as the encryption algorithm. Other
G encryption algorithms require more overhead.

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 or VARCHAR FOR BIT DATA.
- 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 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                                       8 bytes
                                              --------
Encrypted data column length                  16 bytes


Maximum length of non-encrypted data          32 bytes
Number of bytes to the next 8 byte boundary    8 bytes
Overhead                                       8 bytes
                                              --------
Encrypted data column length                  48 bytes
```

Administration of encrypted data: Encrypted data can only be decrypted on servers that support the decryption functions that correspond to the ENCRYPT function. Hence, replication of columns with encrypted data should only be done to servers that support the decryption functions and the same encryption algorithms.

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 = 'Ben123'

INSERT INTO EMP1 (SSN) VALUES ENCRYPT('289-46-8832')
```

- This example explicitly passes the encryption password.

```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT('289-46-8832', 'Ben123')
```

- The hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT('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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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 E is a DECIMAL(10,9) host variable with a value of 3.453789832.

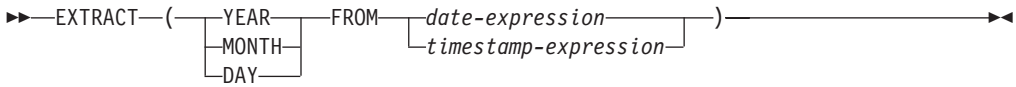
```
SELECT EXP(:E)
   FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 31.62.

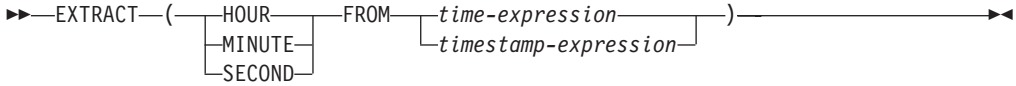
EXTRACT

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 418.

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 313.

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 247.

date-expression
An expression that returns the value of either a built-in date, built-in character string, or built-in graphic string data type. [56]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates, see "String representations of datetime values" on page 66.

timestamp-expression
An expression that returns the value of either a built-in timestamp, built-in character string, or built-in graphic string data type. [56]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 66.

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 281.

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 310.

SECOND
Specifies that the second portion of the time or timestamp expression is returned. The result is identical to the following:

- SECOND(*expression*, 6) when the data type of expression is a TIME value or a string representation of a TIME or TIMESTAMP.
- SECOND(*expression*, *s*) when the data type of expression is a TIMESTAMP(*s*) value.

For more information, see "SECOND" on page 364 and "MICROSECOND" on page 307.

time-expression
An expression that returns the value of either a built-in time, built-in character string, or built-in graphic string data type. [56]

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 with an actual length that is not greater than 255 bytes.[56] For the valid formats of string representations of times, see "String representations of datetime values" on page 66.

timestamp-expression
An expression that returns the value of either a built-in timestamp, built-in character string, or built-in graphic string data type. [56]

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 66.

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 with a TIMESTAMP(*p*) value, the data type of the result is DECIMAL(2+*p*,*p*) where *p* is the fractional seconds precision.
- If SECOND is specified with a TIME value or a string representation of a TIME or TIMESTAMP, the data type of the result is DECIMAL(8,6).

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.

56. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

FLOAT

The FLOAT function returns a floating-point representation of a number.

FLOAT can be specified in place of the DOUBLE_PRECISION and DOUBLE functions. For more information, see "DOUBLE_PRECISION or DOUBLE" on page 265.

FLOOR

The FLOOR function returns the largest integer value less than or equal to *numeric-expression*.

numeric-expression
An expression that returns a value of any built-in numeric data type.

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) return a warning or error.

Examples

- Use the FLOOR function to truncate any digits to the right of the decimal point.

```
SELECT FLOOR(SALARY)
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

  This example returns 52 750.

- 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 (leading zeroes are shown to demonstrate the precision and scale of the result):

```
03.   03.  -04.  -04.
```

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 not-deterministic.

The function has no input parameters. The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and other information that guarantee uniqueness across the relational database. 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.

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 or VARCHAR FOR BIT DATA built-in data type. The data string must have been encrypted using the ENCRYPT 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 the hint parameter was not added to the *encrypted-data* by the ENCRYPT function, the result is the null value.

The CCSID of the result is the default CCSID of the current server.

Examples

- The hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

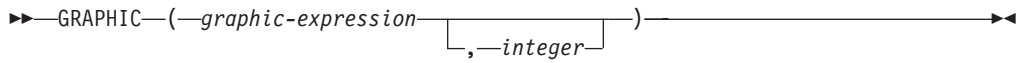
```
INSERT INTO EMP1 (SSN) VALUES ENCRYPT('289-46-8832', 'Pacific', 'Ocean')

SELECT GETHINT(SSN )
  FROM EMP1
```

  The GETHINT function returns the original hint value 'Ocean'.

GRAPHIC

Character to Graphic

Graphic to Graphic



The GRAPHIC function returns a fixed-length graphic-string representation of

- 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.

Character to Graphic

character-expression
An expression that returns a value that is a built-in character-string data type. It cannot be bit data.[57] If the expression is an empty string or the EBCDIC character-string `X'0E0F'`, the result is a single double-byte blank.

If the expression is an empty string or the EBCDIC string `X'0E0F'`, the length attribute of the result is 1. In DB2 for z/OS, if the expression is an empty string, an error is returned. In DB2 for LUW the length attribute of an empty string is 0. Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

Otherwise, the length of the result is the minimum of 127 and the length attribute of *character-expression*.

If the length of *character-expression*, as measured in single-byte characters, is greater than the specified length of the result, as measured in double-byte characters, the result is truncated.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Determining the Graphic Result CCSID: The result of the function is a fixed-length graphic string (GRAPHIC). The CCSID of the result is determined by a mixed data CCSID. Let M denote that mixed data CCSID.

In the following rules, S denotes one of the following:

- If the string expression is a 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 31 for more information.)

57. Although DB2 for z/OS supports storing ASCII data, expression can only be in the EBCDIC encoding scheme.

- 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'
932	301	Japanese ASCII	X'FCFC'
933	834	Korean EBCDIC	X'FEFE'
934	926	Korean ASCII	X'BFFC'
935	837	S-Chinese EBCDIC	X'FEFE'
936	928	S-Chinese ASCII	X'FCFC'
937	835	T-Chinese EBCDIC	X'FEFE'
938	927	T-Chinese ASCII	X'FCFC'
939	300	Japanese EBCDIC	X'FEFE'
942	301	Japanese ASCII	X'FCFC'
943	941	Japanese ASCII	X'FCFC'
944	926	Korean ASCII	X'BFFC'
946	928	S-Chinese ASCII	X'FCFC'
948	927	T-Chinese ASCII	X'FCFC'
949	951	Korean ASCII	X'AFFE'
950	947	T-Chinese ASCII (Big-5)	X'C8FE'
954	13488	Japanese ASCII EUC	X'FFFD'
964	13488	T-Chinese ASCII EUC	X'FFFD'
970	971	Korean ASCII EUC	X'AFFE'
1208	1200	Unicode	Not applicable
1363	1362	Korean ASCII EUC	X'AFFE'
1364	4930	Korean EBCDIC	X'FEFE'
1371	9027	T-Chinese EBCDIC	X'FEFE'
1381	1380	S-Chinese ASCII GB-Code	X'FEFE'
1383	1382	S-Chinese ASCII EUC	X'A1A1'
1386	1385	S-Chinese ASCII EUC	X'A1A1'
1388	4933	S-Chinese EBCDIC	X'FEFE'
1390	16684	Japanese EBCDIC	X'FEFE'
1399	16684	Japanese EBCDIC	X'FEFE'
5026	4396	Japanese EBCDIC	X'FEFE'
5035	4396	Japanese EBCDIC	X'FEFE'

M	Result CCSID	Description	DBCS Substitution Character
5039	1351	Japanese ASCII	X'FFFD'
5307	1351	Japanese ASCII	X'FFFD'

The equivalence of SBCS and DBCS characters depends on M.

Each character of the argument determines a character of the result. Regardless of the character set identified by M, 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 *n*th character of the argument is a DBCS character, the *n*th character of the result is that DBCS character.
- If the *n*th character of the argument is an SBCS character that has an equivalent DBCS character, the *n*th character of the result is that equivalent DBCS character.
- If the *n*th character of the argument is an SBCS character that does not have an equivalent DBCS character, the *n*th character of the result is the DBCS substitution character.
- In the ASCII environment, if the last byte of the argument is a DBCS introducer byte, the last 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.

integer
An integer constant that specifies the length attribute of the result and must be an integer constant between 1 and 127. If the length of *graphic-expression* is less than *integer*, the result is padded with double-byte blanks to the length of the result.

If *integer* is not specified, the length of the result is the minimum of 127 and the length attribute of *graphic-expression*.

If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed.

The CCSID of the result is the same as the CCSID of the first argument.

Note

Syntax alternatives: If the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

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'
```

HEX

The HEX function returns a hexadecimal representation of a value.

expression
An expression that returns a value of any built-in data type that is not XML. A character string must not have a maximum length attribute greater than 16 336. A graphic string must not have a maximum length attribute greater than 8 168.

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 a 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.

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. 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 295.

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 string with a length attribute that is less than one half the product-specific maximum length attribute of CHAR (or GRAPHIC), the result is CHAR with a length attribute that is twice the length attribute of the argument. For more information on the product-specific maximum length, see Chapter 9, "SQL limits," on page 959.
- 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 twice the length attribute of the argument.
  - If the argument is a graphic string, the length attribute of the result is four times the length attribute of the argument.

The length attribute of the result cannot be greater than the product-specific length attribute of CHAR or VARCHAR. See Table 60 on page 964 for more information.

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.[58]

- 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 66.
- 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 137.

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 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
```

58. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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 of the function is 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) for a table containing an identity column. 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 it is replaced by the next assigned value. A new level is initiated when a trigger, function, or 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. [59]

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[60]
- An UPDATE statement
- A ROLLBACK statement with a TO SAVEPOINT clause
- A COMMIT or ROLLBACK statement [59]

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

59. In DB2 for z/OS and DB2 for LUW, a COMMIT or ROLLBACK of a unit of work since the most recent insert operation that assigned a value will also cause the result to be null. In DB2 for i, COMMIT and ROLLBACK do not affect the value.

60. In DB2 for z/OS and DB2 for LUW, an INSERT statement with a subselect does not affect the value.

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 statement be used to obtain the assigned value for an identity column. See "table-reference" on page 470 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 EMPLOYEE2
  (EMPNO INTEGER GENERATED ALWAYS AS IDENTITY,
   NAME CHAR(30),
   SALARY DECIMAL(5,2),
   DEPT SMALLINT)

INSERT INTO EMPLOYEE2
  (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 )
  ```

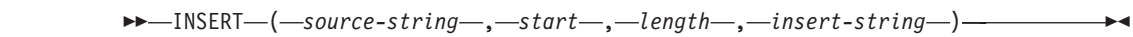
  After the 3 INSERT statements have been processed, table T3 contains the following:

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.

INSERT



The INSERT function 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* must be a built-in character or graphic string expression that is not a CLOB or DBCLOB.

start
An expression that returns a built-in INTEGER or SMALLINT data type. The integer specifies the starting point within the *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 INTEGER or SMALLINT data type. The integer specifies the number of bytes that are to be deleted from the *source-string*, starting at the 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 the *source-string*, starting at the position identified by *start*. The *insert-string* must be a built-in character or graphic string expression that is not a CLOB or DBCLOB. It must be compatible with the *insert-string*. For more information about data type compatibility, see "Assignments and comparisons" on page 77. The actual length of the string must be greater than zero.

G In DB2 for LUW, in a non-Unicode database, when *source-string* is a graphic
G string data type, *insert-string* must also be a graphic string data type; when
G *source-string* is a character string data type, *insert-string* must also be a
G character string data type.

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 95.

G In DB2 for LUW, when using character string and graphic string arguments in a
G Unicode database, the first argument determines whether the result is a character
G string or graphic string data type.

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 is 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 95.

Example

- 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 bytes.

```
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 '.

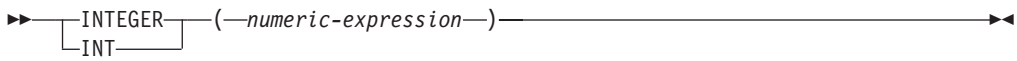
- The following example shows how to insert text after some text. Add 'XX' at the end of string 'ABCABC'. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT INSERT('ABCABC', 7, 0, 'XX')
  FROM SYSIBM.SYSDUMMY1
```

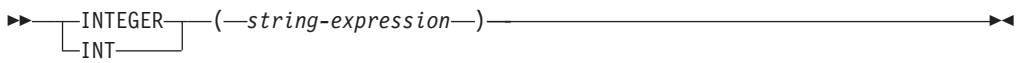
  This example returns 'ABCABCXX '.

INTEGER or INT

Numeric to Integer



String to Integer



The INTEGER function returns an integer representation of:

- A number
- A character-string representation of an integer
- A graphic-string representation of an integer

Numeric to 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 large integer column or variable. If the whole part of the argument is not within the range of large 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 or graphic-string representation of an integer.[61] The expression must not be a CLOB or DBCLOB and must have a length attribute that is not greater than 255 bytes.

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 an integer constant. If the whole part of the argument is not within the range of large integers, an error is returned.

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 for maximal portability. For more information, see "CAST specification" on page 149.

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).

61. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

```
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, or a valid string representation of a date or timestamp.[62] An argument with a character or graphic-string data type must not be a CLOB or DBCLOB and must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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 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 2 438 762.

- 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 2 450 815.

62. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

LAST_DAY

The LAST_DAY scalar 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. [63]

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 or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

The result of the function has the same data type as *expression*, unless *expression* is a string in which case the result is DATE. If the argument can be null, the result can be null; if the 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.

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.

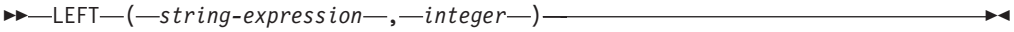
63. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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 300.

LEFT



The LEFT function returns the leftmost *integer* bytes of *string-expression*.

string-expression
An expression that specifies the string from which the result is derived. *string-expression* must be a character string, graphic string, or a binary string with a built-in data type. A substring of *string-expression* is zero or more contiguous bytes of *string-expression*.[64]

integer
An expression that returns a built-in INTEGER or SMALLINT data type. The integer specifies the length of the result. *integer* must be an integer greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *string-expression*.

The *string-expression* is effectively padded on the right with the necessary number of blank characters (or hexadecimal zeroes for binary strings) so that the specified substring of *string-expression* always exists.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *string-expression* and a data type that depends on the data type of *string-expression*:

Data type of *expression*	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
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 *string-expression*.

Example

- Assume the host variable NAME (VARCHAR(50)) has a value of 'KATIE AUSTIN' and the host variable FIRSTNAME_LEN (INTEGER) has a value of 5.

```
SELECT LEFT(:NAME, :FIRSTNAME_LEN)
   FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'KATIE'

64. In EBCDIC environments, if the *string-expression* contains mixed data, the LEFT function operates on a strict byte-count basis. Because LEFT operates on a strict byte-count basis, the result is not necessarily a properly formed mixed data character string.

LENGTH

The LENGTH function returns the length of a value.

expression
An expression that returns a value of any built-in data type that is not XML.

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 integer 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
- 4 for date
- 3 for time
- 7+(p+1)/2 for timestamp(p)

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
   WHERE PROJNO = 'AD3111'
```

  Returns the value 4.

- Assume that PRSTDATE is a column of type DATE.

```
SELECT LENGTH(CHAR(PRSTDATE, EUR))
   FROM PROJECT
   WHERE PROJNO = 'AD3111'
```

  Returns the value 10.

LN

The LN function returns the natural logarithm of a number. The LN and EXP functions are inverse operations.

numeric-expression
An expression that returns a value of one of any built-in numeric data type except for DECFLOAT. The value of the argument must be greater than zero.

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.

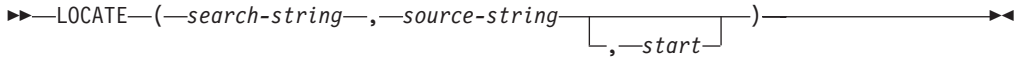
Example

- Assume the host variable 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.

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. The search string must be a character or binary string with an actual length that is no greater than 4000 bytes. It must be compatible with the *source-string*. The expression can be specified by any of the following:

- A constant
- A special register
- A variable
- A scalar function whose arguments are a constant, a special register, or a variable (though nested function invocations cannot be used)
- An expression that concatenates any of the above

source-string
An expression that specifies the source string in which the search is to take place. The source string must be a character or binary string with an actual length that is no greater than 4000 bytes. The expression can be specified by any of the following:

- A constant
- A special register
- A variable
- A scalar function whose arguments are a constant, a special register, or a variable (though nested function invocations cannot be used)
- A column name
- An expression that concatenates any of the above

start
An expression that specifies the position within *source-string* at which the search is to start. It must be an integer that is greater than or equal to zero.

If *start* is specified, the function is similar to:

```
POSSTR( SUBSTR(source-string, start), search-string ) + start - 1
```

If *start* is not specified, the function is equivalent to:

```
POSSTR( source-string, search-string )
```

For more information, see “POSSTR” on page 335.

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.

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 *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.[65]

Note

Syntax alternatives: The POSSTR function is similar to the LOCATE function. For more information, see "POSSTR" on page 335.

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
```

65. This includes the case where the *search-string* is longer than the *source-string*.

LOG10

The LOG10 function returns the common logarithm (base 10) of a number.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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 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.

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 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.

string-expression
An expression that specifies the string to be converted. *String-expression* must be a character or graphic string that is not a CLOB or DBCLOB.

G In DB2 for LUW, a graphic string is only allowed in a Unicode database. If a
G supplied argument is a graphic string, it is first converted to a character string
G before the function is executed.

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 can be specified in place of LOWER.

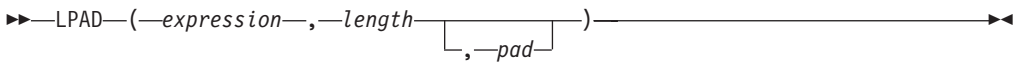
Example

- 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 source string. The expression must return a value that is a built-in string data type that is not a LOB.

length
An expression that specifies the length of the result. The value must be zero or a positive integer constant that is less than or equal to *n*, where *n* is the maximum length of the result data type. See Chapter 9, "SQL limits," on page 959 for more information.

pad
An expression that specifies the string with which to pad. The expression must return a value that is a built-in string data type that is not a LOB.

If *pad* is not specified, the pad character is set as follows:

- SBCS blank character if string-expression is a character string.
- DBCS blank character if string-expression is a graphic string.[66]

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 77.

The data type of the result depends on the data type of *expression*:

Data type of *expression*	**Data Type of the Result for LPAD**
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC

The length attribute of the result depends on *length*. If *length* is greater than zero, the length attribute of the result is *length*. If *length* is zero, the length attribute of the result is 1.

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.
- If *length* is greater than the actual length of *expression*, the actual length is *length*.

66. 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'.

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 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
```

- 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
```

- 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
```

- 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
```

- 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 from the beginning of a string expression.

string-expression
An expression that returns a value that is a built-in character string data type or graphic string data type. The argument must not be a CLOB or DBCLOB. The characters that are interpreted as leading blanks depend on the data type and the encoding scheme of the data:

- If the argument is a DBCS graphic string, then the leading DBCS blanks are removed.
- If the first argument is a Unicode graphic string, then the leading Unicode blanks are removed
- Otherwise, leading 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
GRAPHIC or VARGRAPHIC	VARGRAPHIC

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 blanks removed. If all characters are removed, the result is an empty string.

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 same as that of the string.

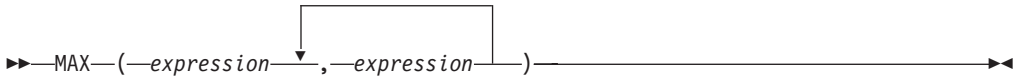
Example

- Assume the host variable HELLO of type CHAR(9) has a value of ' Hello'.

```
SELECT LTRIM(:HELLO)
   FROM SYSIBM.SYSDUMMY1
```

  Results in: 'Hello'.

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. For more information on data type compatibility, see "Assignments and comparisons" on page 77.

expression
An expression that returns a value of any built-in data type other than BLOB, CLOB, DBCLOB, or XML or a user-defined data type not based on a BLOB, CLOB, or DBCLOB.[67] The expression must have a length attribute that is not greater than 255 bytes.

G DB2 for z/OS does not support user-defined data type arguments.

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 91.

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.

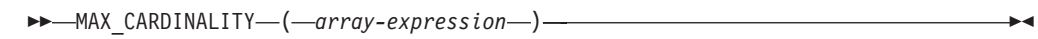
- Assume the host variable M1 is a CHARACTER(2) host variable with a value of 'AA', host variable M2 is a CHARACTER(3) host variable with a value of 'AA ', and host variable M3 is a CHARACTER(4) host variable with a value of 'AA A'.

```
SELECT MAX(:M1,:M2,:M3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'AA A'.

67. 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.

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 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 can be null.

G In DB2 for LUW, the result of the function is INTEGER.

Example

Assume that array type PHONENUMBERS and array variable RECENT_CALLS are defined as follows:

```
CREATE TYPE PHONENUMBERS AS INTEGER ARRAY[50];
DECLARE RECENT_CALLS PHONENUMBERS;
```

The following statement sets LIST_SIZE to the maximum cardinality with which RECENT_CALLS was defined.

```
SET LIST_SIZE = MAX_CARDINALITY(RECENT_CALLS)
```

After the statement executes, LIST_SIZE contains 50.

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. [68]

- 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 66.
- 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 137.

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 string representation of a timestamp:

  The result is the microsecond part of the value, which is an integer between 0 and 999 999.

  If the precision of the timestamp exceeds 6, the value is truncated.
- If the argument is a duration:

  The result is the microsecond part of the value, which is an integer between -999 999 and 999 999. 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)
```

68. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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, or a valid string representation of a time or timestamp. [69] An argument with a character or graphic-string data type must not be a CLOB or DBCLOB and must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 66.

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

- 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 47 410. 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 47 410 seconds ((3600 * 13) + (60 * 10) + 10).

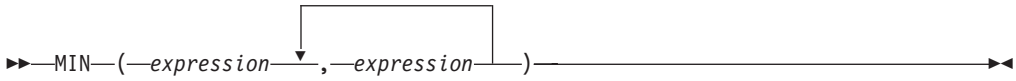
- Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

```
SELECT MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00')
  FROM SYSIBM.SYSDUMMY1
```

  This example returns 86 400 and 0. Although these two values represent the same point in time, different values are returned.

69. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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. For more information on data type compatibility, see "Assignments and comparisons" on page 77.

expression
An expression that returns a value of any built-in data type other than BLOB, CLOB, DBCLOB, or XML or a user-defined data type not based on a BLOB, CLOB, or DBCLOB.[70] The expression must have a length attribute that is not greater than 255 bytes.

G DB2 for z/OS does not support user-defined data type arguments.

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 91.

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 'AA ', and host variable M3 is a CHARACTER(4) host variable with a value of 'AA A'.

```
SELECT MIN(:M1,:M2,:M3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'AA '.

70. 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.

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.[71]

- 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 66.
- 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 137.

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 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.

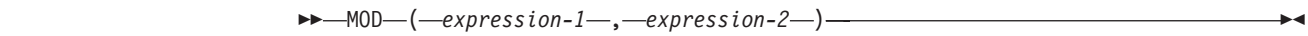
Example

- 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
```

71. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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 numeric, character-string, or graphic-string data type. The expression must not be a CLOB or DBCLOB and must have a length attribute that is not greater than 255 bytes. A string argument is cast to double-precision floating point or decimal floating point before evaluating the function..

expression-2
An expression that returns a value of any numeric, character-string, or graphic-string data type. The expression must not be a CLOB and must have a length attribute that is not greater than 255 bytes. A string argument is cast to double-precision floating point or decimal floating point before evaluating the function.. *expression-2* cannot be zero unless at least one of the arguments 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, the data type of the result is large integer. In DB2 for LUW if both arguments are small integers, the result is a small integer.
- If 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 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 floating point and the other operand is not decimal floating-point, the data type of the result is double-precision floating point.

  The operation is performed in floating point. If necessary, the operands are first converted to double precision floating-point numbers. For example, an operation that involves a floating-point number and either an integer or a decimal number is performed with a temporary copy of the integer or 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, the data type of the result is DECFLOAT(34). If the second argument is zero, the result is NaN and a warning is returned. [72]

  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 133 for more information.

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.

72. In DB2 for i the warning is only returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

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.[73]

- 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.
- 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 137.

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 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
```

73. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

MONTHNAME

The MONTHNAME function returns a mixed case character string containing the name of the month (e.g. January) for the month portion of the argument. [74] The
G name of the month depends on the national language (locale). In DB2 for z/OS the
G name of the month is returned in English.

expression
An expression that returns a value of one of the following built-in data types: a date or a character string.

If *expression* is a character string, it must not be a CHAR or CLOB and its value must be a valid string representation of a date in ISO format with an actual length not larger than 10. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

G The result of the function is VARCHAR(100). In DB2 for z/OS, the result of the
G function is VARCHAR(9). 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.

Example

- Assume that the language used is US English.

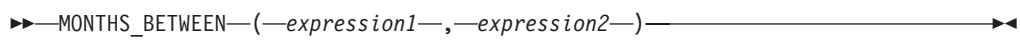
```
SELECT MONTHNAME('2003-01-02')
  FROM SYSIBM.SYSDUMMY1
```

Results in 'January'.

74. In DB2 for z/OS, installation job DSNTEJ2U must be run to use this function.

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. [75]

If *expression1* 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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. [75]

If *expression2* 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

If *expression1* represents a date that is equal to or 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 or timestamps 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 precision 12, 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

75. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

- The following table contains additional examples:

Table 30. Additional examples using MONTHS_BETWEEN

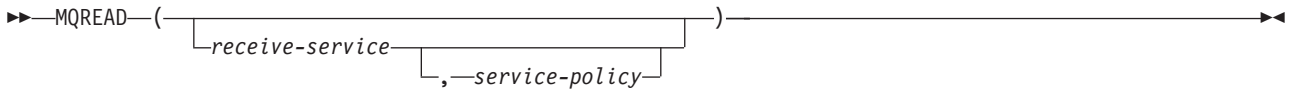
Value for argument *e1*	**Value for argument** *e2*	**Value returned by MONTHS_BETWEEN (***e1,e2***)**	**Value returned by DECIMAL(ROUND(MONTHS_BETWEEN(** *e1,e2***)*31,2), 15, 2)**	**Comment**
2005-02-02	2005-01-01	1.032258064516129	32.00	
2007-11-01-09.00.00.00000	2007-12-07-14.30.12.12345	-1.200945386592741	-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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

The result of the function is a varying-length string with a length attribute of 4000. See Table 63 on page 967 for more information.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 must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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).

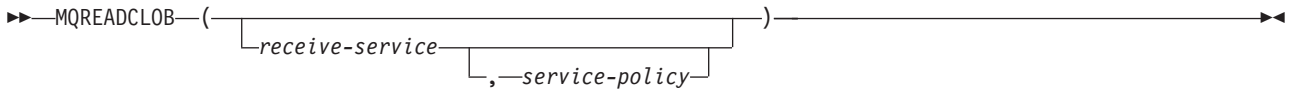
```
SELECT MQREAD ('MYSERVICE')
 FROM SYSIBM.SYSDUMMY1
```

- 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
```

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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

If *service-policy* is not specified or the null value, DB2.DEFAULT.POLICY is used.

The result of the function is a CLOB with a length attribute of 1M. See Table 63 on page 967 for more information. 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 must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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).

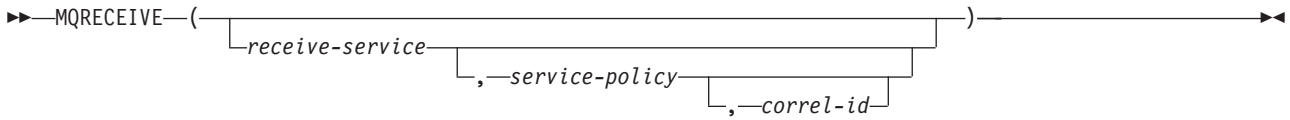
```
SELECT MQREADCLOB ('MYSERVICE')
 FROM SYSIBM.SYSDUMMY1
```

- 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
```

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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

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.

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 4000. See Table 63 on page 967 for more information. 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 must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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".

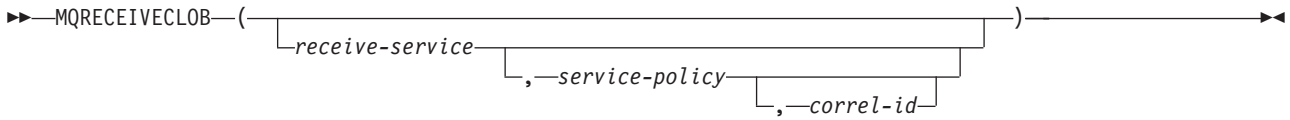
```
SELECT MQRECEIVE ('MYSERVICE','MYPOLICY')
 FROM SYSIBM.SYSDUMMY1
```

- 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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

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.

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 1M. See Table 63 on page 967 for more information. 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 must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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".

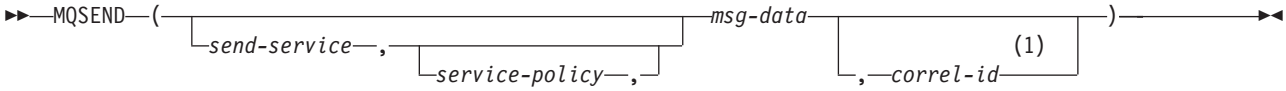
```
SELECT MQRECEIVECLOB ('MYSERVICE','MYPOLICY')
 FROM SYSIBM.SYSDUMMY1
```

- 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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

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 1M. Otherwise, the value must not be longer than 4000. See Table 63 on page 967 for more information. 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.

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 must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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
```

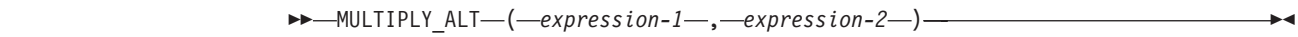
- This example sends the string "Testing 678" to the service "MYSERVICE", using the policy "MYPOLICY", with correlation identifier "TEST3".

```
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 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 31.

expression-1
An expression that returns a value of any built-in exact numeric data type (DECIMAL, NUMERIC, BIGINT, INTEGER, or SMALLINT).

expression-2
An expression that returns a value of any built-in exact numeric data type (DECIMAL, NUMERIC, BIGINT, INTEGER, or SMALLINT). *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(31, p+p')
- The scale is:
  - 0 if the scale of both arguments is 0
  - MIN(31, s+s') if p+p' is less than or equal to 31
  - MIN(31, MAX(MIN(3, s+s'), 31-(p-s+p'-s'))) if p+p' is greater than 31.

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 desired and the sum of the precisions exceeds 31. 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:

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)
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)

Example

- 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(98765432109876543210987.654,5.43210987)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 536504678578875294857887.5277415.

  Note that the complete product of these two numbers is 536504678578875294857887.52774154498, 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 after the day named by the second 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. [76]

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 or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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.[76] 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. The name and abbreviation of the day depends on the national language (locale). In DB2 for z/OS the name of the day is returned in English. 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 blanks must not be specified in *string-expression*. 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(6). 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.

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')
```

76. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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, 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 a DECFLOAT data type.

If the argument is a special value then the general rules for arithmetic operations apply. See"General arithmetic operation rules for DECFLOAT" on page 133 for more information.

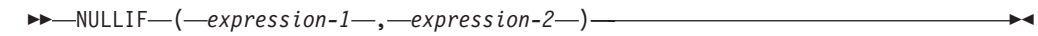
The result of the function is a DECFLOAT(16) value if the data type of expression is DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. If the argument can be null, the result can be null; if the argument is null, the result is the null value

Examples

The following examples demonstrate the values that will be returned when the function is used:

```
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 (DECFLOAT(-NAN))        = -NAN
NORMALIZE_DECFLOAT (DECFLOAT(-INFINITY))   = -INFINITY
```

NULLIF



The NULLIF function returns a null value if the arguments compare equal, otherwise it returns the value of the first argument.

expression-1
An expression that returns a value of any built-in data types other than a BLOB, CLOB, DBCLOB, or XML.

expression-2
An expression that returns a value of any built-in data types other than a BLOB, CLOB, DBCLOB, or XML.

The arguments must be compatible and comparable built-in data types. Character-string arguments are compatible and comparable with datetime values. For more information on data type compatibility, see "Assignments and comparisons" on page 77.

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.

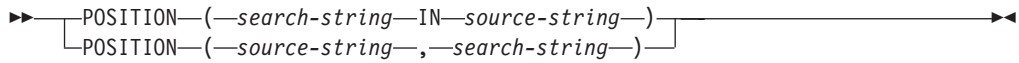
Example

- 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.

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 in characters of the *source-string*. See the related functions, "LOCATE" on page 297 and "POSSTR" on page 335.

search-string
: An expression that specifies the string that is to be searched for. *search-string* must return a value that is a built-in string data type. The expression must have an actual length that is no greater than 4000 bytes. It must be compatible with the *source-string*. The expression can be specified by any of the following:
  - A constant
  - A special register
  - A variable
  - A scalar function whose arguments are a constant, a special register, or a variable (though nested function invocations cannot be used)
  - An expression that concatenates any of the above

source-string
: An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is a built-in string data type. The expression can be specified by any of the following:
  - A constant
  - A special register
  - A variable (including a LOB locator variable)
  - A scalar function whose arguments are a constant, a special register, or a variable (though nested function invocations cannot be used)
  - A column name
  - An expression that concatenates any of the above

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, 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 *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.[77]

G DB2 for z/OS, does not support the POSITION(*search-string* IN *source-string*) form
G of the function. In DB2 for z/OS and DB2 for LUW a third argument is required.
G CODEUNITS32 should be specified for the third argument.

Examples

- Select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD' within the NOTE_TEXT column for all rows in the IN_TRAY table that contain that string.

```
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.

77. 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 in bytes of the *source-string*.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is a built-in string data type. The expression can be specified by any of the following:

- A constant
- A special register
- A variable (including a LOB locator variable)
- A scalar function whose arguments are a constant, a special register, or a variable (though nested function invocations cannot be used)
- A column name
- An expression that concatenates any of the above
- A CAST specification whose arguments are any of the above

search-string
An expression that specifies the string that is to be searched for. *search-string* must return a value that is a built-in string data type. The expression must have an actual length that is no greater than 4000 bytes. It must be compatible with the *source-string*. The expression can be specified by any of the following:

- A constant
- A special register
- A variable
- A scalar function whose arguments are a constant, a special register, or a variable (though nested function invocations cannot be used)
- An expression that concatenates any of the above
- A CAST specification whose arguments are any of the above

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.[78] 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.

78. 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 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 *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.[79]

Note

Syntax alternatives: The LOCATE and POSITION functions are similar to the POSSTR function. For more information, see "LOCATE" on page 297.

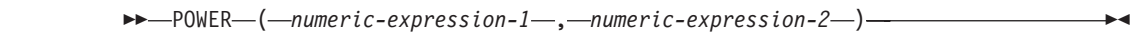
Example

- 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
```

79. 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. [80]

numeric-expression-1
An expression that returns a value of any built-in numeric data type.

If the value of *numeric-expression-1* is equal to zero, then *numeric-expression-2* must be greater than or equal to zero. If both arguments are 0, the result is 1. If the value of *numeric-expression-1* is less than zero, then *numeric-expression-2* must be an integer value.

numeric-expression-2
An expression that returns a value of any built-in numeric data type.

If either argument is decimal floating-point, both arguments are converted to DECFLOAT(34). Otherwise, the arguments are converted to double-precision floating-point for processing by the function.

If the data type of either 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. In DB2 for z/OS and DB2 for LUW the result is INTEGER if both arguments are either INTEGER or SMALLINT. 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.
- 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.
- If either argument is sNaN or -sNaN, a warning or error is returned.

Example

- Assume the host variable HPOWER is an integer with value 3.

```
SELECT POWER(2,:HPOWER)
   FROM SYSIBM.SYSDUMMY1
```

  Returns the value 8.

80. The result of the POWER function is exactly the same as the result of exponentiation: *numeric-expression-1* ** *numeric-expression-2*.

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 data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

expression-2
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.

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 103 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. [81]
- 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.
- If both arguments are Infinity (positive or negative), then Infinity (positive or negative) is returned.

81. In DB2 for i the warning is only returned if *YES is specified for the SQL_DECFLOAT_WARNINGS query option.

- If one argument is Infinity (positive or negative) and the other argument is not Infinity (positive or negative), then NaN is returned with a warning.[81]

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      -- Warning
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.

```
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.[82]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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.

82. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

RADIANS

The RADIANS function returns the number of radians for an argument that is expressed in degrees.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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.

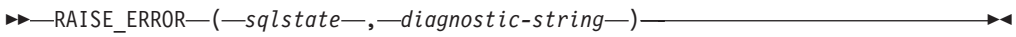
Example

- 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 error condition. The RAISE_ERROR function always returns NULL with an undefined data type.

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 non-accented upper case 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 of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it will be truncated.

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 cannot be used with CASE expressions.

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 between 0 and 1.

numeric-expression
An expression that returns a value of a built-in INTEGER or SMALLINT data type with a value between 0 and 2 147 483 646. If an expression is specified, it is used as the seed value.

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, other than zero, 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
G numbers is produced each time the query is executed. The result with a seed value
G of zero is product-specific.

RAND is a non-deterministic function.

Examples

- 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 desired 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 of a number.

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 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.

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

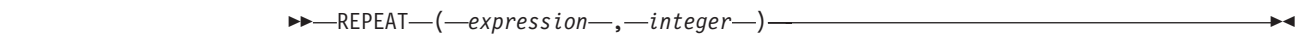
Syntax alternatives: The CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

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
```

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 may be any built-in character or graphic string expression that is not a CLOB or DBCLOB. The actual length of the string must be less or equal to 4000.

G In DB2 for LUW, a graphic string is only allowed in a Unicode database. If a G supplied argument is a graphic string, it is first converted to a character string G before the function is executed.

integer
An expression that returns a built-in 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:
| • VARCHAR if *expression* is a character string
| • VARGRAPHIC if *expression* is graphic string
G In DB2 for LUW the result is VARCHAR if *expression* is graphic string.

| If *integer* is a constant, the length attribute of the result is the length attribute of | *expression* times *integer*. Otherwise, the length attribute depends on the data type of | the result:
| • 4000 for VARCHAR
| • 2000 for VARGRAPHIC

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 *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 *expression*.[83]

Examples

- Repeat 'abc' two times to create 'abcabc'.

```
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.

83. If the value of *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 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 character or graphic string expression that is not a CLOB or DBCLOB. The actual length of the string must be greater than zero and less or equal to 4000.

search-string
An expression that specifies the string to be removed from the source string. The *search-string* must be a built-in character or graphic string expression that is not a CLOB or DBCLOB. The actual length of the string must be greater than zero and less or equal to 4000.

replace-string
An expression that specifies the replacement string. The *replace-string* must be a built-in character or graphic string expression that is not a CLOB or DBCLOB. The actual length of the string must be less or equal to 4000.

source-string, *search-string*, and *replace-string* must be compatible. For more information about data type compatibility, see "Assignments and comparisons" on page 77.

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 95.

G In DB2 for LUW, the data type of the result is the data type of the first argument.

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)
  ```
- G In DB2 for LUW:
  - G If the data type of the result is VARCHAR and L1 is greater than 4000, the G length attribute is no larger than 4000.
  - G If the data type of the result is VARGRAPHIC and L1 is greater than 2000, the G length attribute is no larger than 2000.

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 95.

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 row identifier (RID) 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 111.

The *table-designator* must not identify a *table-function*, a *collection-derived-table*or a *data-change-table-reference*.

If *table-designator* specifies a view or a nested table expression, the RID function returns the RID of the base table of the view or nested table expression. The specified view or nested table expression must contain only one base table in its outer subselect.

The *table-designator* 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, VALUES clause, or a *table-function*. 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.

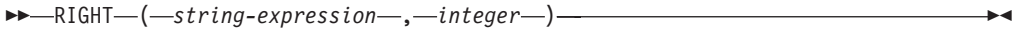
The result of the function is BIGINT. The result can be null.

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 *string-expression*.

If *string-expression* is a character string, the result is a character string, and each character is one byte. If *string-expression* is a binary string, the result is a binary string, and each character is one byte.

string-expression
An expression that specifies the string from which the result is derived. *string-expression* must be a character string, graphic string, or a binary string with a built-in data type. A substring of *string-expression* is zero or more contiguous bytes of *string-expression*.[84]

integer
An expression that returns a built-in INTEGER or SMALLINT 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 *string-expression*.

The *string-expression* is effectively padded on the right with the necessary number of blank characters (or hexadecimal zeroes for binary strings) so that the specified substring of *string-expression* always exists.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *string-expression* and a data type that 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
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 *string-expression*.

Examples

- Assume that host variable ALPHA has a value of 'ABCDEF'. The following statement:

```
SELECT RIGHT(:ALPHA, 3)
  FROM SYSIBM.SYSDUMMY1
```

84. In EBCDIC environments, if the *string-expression* contains mixed data, the RIGHT function operates on a strict byte-count basis. Because RIGHT operates on a strict byte-count basis, the result is not necessarily a properly formed mixed data character string.

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
```

ROUND

The ROUND function returns *numeric-expression–1* rounded to *numeric-expression–2* places to the right or left of the decimal point.

numeric-expression–1
An expression that returns a value of any built-in numeric data type.

If *numeric-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.

numeric-expression–2
An expression that returns a value of a built-in INTEGER or SMALLINT data type.

If *numeric-expression–2* is positive, *numeric-expression–1* is rounded to the *numeric-expression–2* number of places to the right of the decimal point.

If *numeric-expression–2* is negative, *numeric-expression–1* is rounded to 1 + (the absolute value of *numeric-expression–2*) number of places to the left of the decimal point. If the absolute value of *numeric-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 *numeric-expression–1* is positive, a digit value of 5 is rounded to the next higher positive number. If *numeric-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 *numeric-expression–1* is DECIMAL or NUMERIC and the precision is less than 31. 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(31,2) will result in DECIMAL(31,2).

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
```

  This example returns (leading zeroes are shown to demonstrate the precision and scale of the result):

```
0873.730   0873.700  0874.000  0870.000  0900.000  1000.000  0000.000
```

- Calculate both positive and negative numbers.

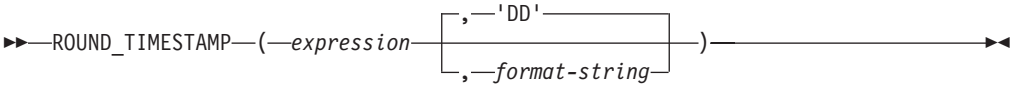
```
SELECT ROUND( 3.5, 0),
       ROUND( 3.1, 0),
       ROUND(-3.1, 0),
       ROUND(-3.5, 0)
  FROM TABLEX
```

This example returns:

```
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. [85]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 66.

format-string
An expression that returns a built-in character string data type or graphic string data type that is not a CLOB of DBCLOB. The actual length of the expression must not be greater than 254 bytes.[85] *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 not removed from the string. The resulting substring must be a valid format element for a timestamp.

Allowable values for *format-string* are listed in Table 31.

The result of the function is a timestamp with a timestamp precision of:

- *p* when the data type of expression is TIMESTAMP(*p*)
- 6 otherwise.

If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Table 31. 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

85. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

Table 31. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models (continued)

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
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
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

Table 31. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models (continued)

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
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
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 417 for details.			

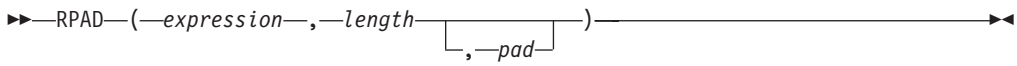
Example

- 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.

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 source string. The expression must return a value that is a built-in string data type that is not a LOB.

length
An expression that specifies the length of the result. The value must be zero or a positive integer constant that is less than or equal to *n*, where *n* is the maximum length of the result data type. See Chapter 9, "SQL limits," on page 959 for more information.

pad
An expression that specifies the string with which to pad. The expression must return a value that is a built-in string data type that is not a LOB.

If *pad* is not specified, the pad character is set as follows:

- SBCS blank character if string-expression is a character string.
- DBCS blank character if string-expression is a graphic string.[86]

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 77.

The data type of the result depends on the data type of *expression*:

Data type of *expression*	**Data Type of the Result for LPAD**
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC

The length attribute of the result depends on *length*. If *length* is greater than zero, the length attribute of the result is *length*. If *length* is zero, the length attribute of the result is 1.

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.
- If *length* is greater than the actual length of *expression*, the actual length is *length*.

86. 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'.

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 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...........
```

- 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.
```

- 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...........
```

- 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
```

- 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

|
```

RTRIM

The RTRIM function removes blanks from the end of a string expression.

string-expression
An expression that returns a value of any built-in character string data type or graphic-string data type, that is not a CLOB or DBCLOB. The characters that are interpreted as trailing blanks depend on the data type and the encoding scheme of the data:

- 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 Unicode 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
GRAPHIC or VARGRAPHIC	VARGRAPHIC

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 blanks 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.

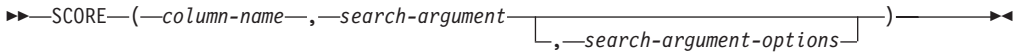
Example

- Assume the host variable HELLO of type CHAR(9) has a value of 'Hello '.

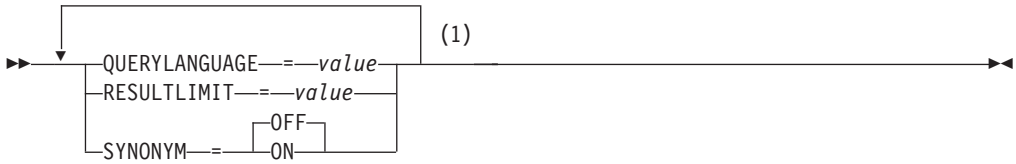
```
SELECT RTRIM(:HELLO)
    FROM SYSIBM.SYSDUMMY1
```

Results in: 'Hello'.

SCORE



search-argument-options



Notes:

1 The same clause must not be specified more than once.

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.

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 is not a CLOB or DBCLOB 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 4 096 Unicode characters. The value is converted to Unicode before it is used to search the text search index. The value 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 the product documentation.

search-argument-options
A character string or graphic string constant that specifies the search argument options to use for the search. The options that can be specified as part of the *search-argument-options* are:

QUERYLANGUAGE = *value*
Specifies the language value. The value must be one 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 the product documentation.

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 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.

Note

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.

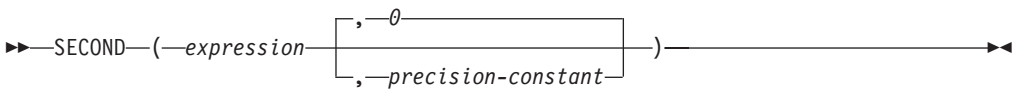
Example

- 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 with optional fractional seconds.

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.[87]

- 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 66.
- 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 137.

precision-constant
An integer constant representing the number of fractional seconds. The value must be in the range 0 through 12.

The result of the function with a single argument is a large integer. The result of the function with two arguments is DECIMAL(2+*s*,*s*) where *s* is the value of the *precision-constant*. 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 other rules depend on the data type of the argument:

- If the argument is a time, a timestamp, or a valid string representation of a time or timestamp:

  If only one argument is specified, the result is the seconds part of the value (0 to 59).

  If both arguments are specified, the result is the seconds part of the value (0 to 59) and *precision-constant* digits of the fractional seconds part of the value. If there are no fractional seconds in the value, then zeroes are returned.
- If the argument is a time duration or timestamp duration:

  If only one argument is specified, the result is the seconds part of the value (-99 to 99). A nonzero result has the same sign as the argument.

  If both arguments are specified, the result is the seconds part of the value (-99 to 99) and *precision-constant* digits of the fractional seconds part of the value. If there are no fractional seconds in the value, then zeroes are returned. 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
```

87. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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
   WHERE SOURCE = 'BADAMSON'
```

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

numeric-expression
An expression that returns a value of any built-in numeric data type, except DECIMAL(31,31).

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
G data type of DECIMAL(5,5) will result in DECIMAL(6,5). In DB2 for LUW the
G result is DOUBLE if the argument is DECIMAL or REAL.

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 50 000.

```
SELECT SIGN(:PROFIT)
  FROM SYSIBM.SYSDUMMY1
```

  This example 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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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

Numeric to Smallint

String to Smallint

The SMALLINT function returns an integer representation of:

- A number
- A character-string representation of an integer
- A graphic-string representation of an integer

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 or graphic-string representation of an integer.[88] The expression must not be a CLOB or DBCLOB and must have a length attribute that is not greater than 255 bytes.

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 an integer constant. If the whole part of the argument is not within the range of small integers, an error is returned.

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 for maximal portability. For more information, see "CAST specification" on page 149.

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
```

88. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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.

string-expression
An expression that returns a built-in character-string or graphic-string data type, that is not a CLOB or DBCLOB. The argument cannot be a binary string.[89]

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 263.

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
```

89. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

SPACE

The SPACE function returns a character string that consists of the number of SBCS blanks that the argument specifies.

numeric-expression
An expression that returns a value that is a built-in INTEGER or SMALLINT data type. The integer specifies the number of SBCS blanks for the result, and it must be between 0 and 4000. If *numeric-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 *numeric-expression* is a constant, the length attribute of the result is the constant. Otherwise, the length attribute of the result is 4000. In DB2 for LUW the length attribute is always 4000. The actual length of the result is the value of *numeric-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 of the result is the default CCSID at the current server.

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.

numeric-expression
An expression that returns a value of any built-in numeric data type. The value of *numeric-expression* must be greater than or equal to zero. The argument is converted to double-precision floating point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n). Otherwise, 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

Results involving DECFLOAT special values: 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 133 for more information.

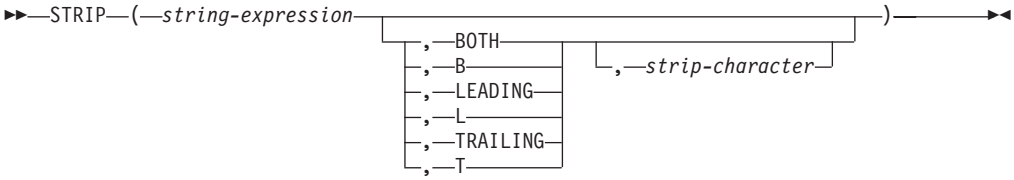
Example

- 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.

STRIP



The STRIP function removes blanks or another specified character from the end, the beginning, or both ends of a string expression.

The second argument, if specified, indicates whether characters are removed from the end or beginning of the string. If the second argument is not specified, then the characters are removed from both the end and the beginning of the string.

string-expression
An expression that returns a built-in character-string or graphic-string data type, that is not a CLOB or DBCLOB. The argument cannot be a binary-string.

strip-character
The third argument, if specified, is a single-character constant that indicates the SBCS or DBCS character that is to be removed. If *string-expression* is a DBCS graphic string, the third argument must be a graphic constant consisting of a single DBCS character. If the third argument is not specified then:

- If *string-expression* is a DBCS graphic string, then the default strip character is a DBCS blank.
- If *string-expression* is a Unicode graphic string, then the default strip character is a UTF-16 or UCS-2 blank.
- If *string-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.

The data type of the result depends on the data type of *string-expression*:

Data type of *expression*	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC

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.

Examples

- Assume the host variable HELLO of type CHAR(9) has a value of ' Hello '.

```
SELECT STRIP(:HELLO), STRIP( :HELLO, TRAILING)
  FROM SYSIBM.SYSDUMMY1
```

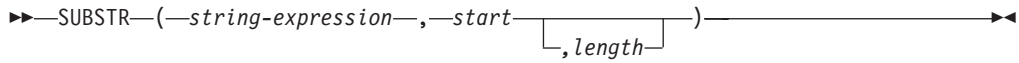
Results in 'Hello' and ' Hello' respectively.

- Assume the host variable BALANCE of type CHAR(9) has a value of '000345.50'.

```
SELECT STRIP( :BALANCE, L, '0')
  FROM SYSIBM.SYSDUMMY1
```

Results in: '345.50'

SUBSTR



The SUBSTR function returns a substring of a string.

string-expression
An expression that specifies the string from which the result is derived. *string-expression* must be a built-in character, graphic, or binary string. If *string-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 *string-expression* is zero or more contiguous characters of *string-expression*. If *string-expression* is a graphic string, a character is a DBCS or Unicode character. If *string-expression* is a character string or binary string, a character is a byte. The SUBSTR function accepts mixed data strings. However, because SUBSTR operates on a strict byte-count basis for character strings, the result will not necessarily be a properly formed mixed data string.

start
An expression that specifies the position within *string-expression* of the first character (or byte) of the result. The expression must return a value that is a built-in INTEGER or SMALLINT data type. The value must be greater than zero and less than or equal to the length attribute of *string-expression*.

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 INTEGER or SMALLINT data type. The value must be greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *string-expression* - *start* + 1. It must not, however, be the integer constant 0.

If *length* is explicitly specified, *string-expression* is effectively padded on the right with the necessary number of blank characters so that the specified substring of *string-expression* always exists. Hexadecimal zeroes are used as the padding character when *string-expression* is a binary string.

If *string-expression* is a fixed-length string, omission of *length* is an implicit specification of LENGTH(*string-expression*) - *start* + 1, which is the number of characters (or bytes) from the character (or byte) specified by *start* to the last character (or byte) of *string-expression*. If *string-expression* is a varying-length string, omission of *length* is an implicit specification of the greater of zero or LENGTH(*string-expression*) - *start* + 1. If the resulting length is zero, the result is the empty string.

The data type of the result depends on the data type of *string-expression*:

Data type of *string-expression*	Data Type of the Result for SUBSTR
CHAR or VARCHAR	CHAR, if: • *length* is explicitly specified by an integer constant that is less than the product-specific maximum of a character-string. See Table 60 on page 964 for more information. • *length* is not explicitly specified, but *string-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 less than the product-specific maximum of a graphic-string. See Table 60 on page 964 for more information. • *length* is not explicitly specified, but *string-expression* is a fixed-length string and *start* is an integer constant. VARGRAPHIC, in all other cases.
DBCLOB	DBCLOB
BLOB	BLOB

If *string-expression* is not a LOB, the length attribute of the result depends on *length*, *start*, and the attributes of *string-expression*.

- If *length* is explicitly specified by an integer constant, the length attribute of the result is *length*.
- If *length* is not explicitly specified, but *string-expression* is a fixed-length string and *start* is an integer constant, the length attribute of the result is LENGTH(*string-expression*) - *start* + 1.

In all other cases, the length attribute of the result is the same as the length attribute of *string-expression*. (Remember that if the actual length of *string-expression* is less than the value for *start*, the actual length of the substring is zero.)

If any argument of the SUBSTR function 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 *string-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'

```
SELECT SUBSTR(:NAME, :SURNAME_POS, 1)
   FROM SYSIBM.SYSDUMMY1
```

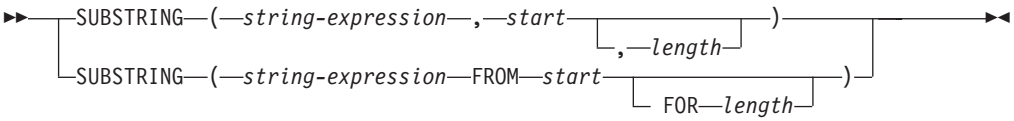
  Returns the value 'A'.

- 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.

string-expression
An expression that specifies the string from which the result is derived.

string-expression must be any built-in string data type. If *string-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 *string-expression* is zero or more contiguous characters of *string-expression*. If *string-expression* is a graphic string, a character is a DBCS or Unicode graphic character. If *string-expression* is a character string, a character is a character that may consist of one or more bytes. If *string-expression* is a binary string, a character is a byte.

start
An expression that specifies the position within *string-expression* of the first character (or byte) of the result. The expression must return a value that is a built-in INTEGER or SMALLINT data type. The value must be greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *string-expression* - *start* + 1. It must not, however, be the integer constant 0.

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 INTEGER or SMALLINT data type. The value must be greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *string-expression* - *start* + 1. It must not, however, be the integer constant 0.

If *string-expression* is a fixed-length string, omission of *length* is an implicit specification of LENGTH(*string-expression*) - *start* + 1, which is the number of characters (or bytes) from the *start* character (or byte) to the last character (or byte) of *string-expression*. If *string-expression* is a varying-length string, omission of *length* is an implicit specification of zero or LENGTH(*string-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 *string-expression*:

Data type of *string-expression*	**Data Type of the Result for SUBSTRING**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BLOB	BLOB

The length attribute of the result is the same as the length attribute of *string-expression*.

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 *string-expression*.

G In DB2 for z/OS and DB2 for LUW a fourth argument is required. CODEUNITS32
G should be specified for the fourth argument.

G DB2 for z/OS does not support the form of SUBSTRING that contains the FROM
G keyword.

Examples

- Select all rows from the PROJECT table for which the project name (PROJNAME) starts with the word 'OPERATION '. The space at the end of the constant is necessary to preclude initial words such as 'OPERATIONS'.

```
SELECT * FROM PROJECT
  WHERE SUBSTRING(PROJNAME, 1, 10) = 'OPERATION '
```

- 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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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.

numeric-expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT.

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. [90]

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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 string:

  The result is the time or time part of the timestamp represented by the 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 for maximal portability. For more information, see "CAST specification" on page 149.

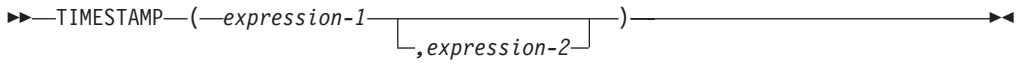
Example

- 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
```

90. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

TIMESTAMP



The TIMESTAMP function returns a timestamp from its argument or arguments.

expression-1 **and** *expression-2*

The rules for the arguments depend on whether a second argument is specified and the data type of the second argument.

- 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 a date, timestamp, a character string, or a graphic string.[91] 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 date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 66.

  – A string with an actual length of 14 that represents a valid date and time in the form yyyyxxddhhmmss, where yyyy is year, xx is month, dd is day, hh is hour, mm is minute, and ss is seconds.

  – 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 275.

- If both arguments are specified:

  – If the data type of *expression-2* is not an integer:

    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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates, see "String representations of datetime values" on page 66.

    expression-2 must be 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 date with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times, see "String representations of datetime values" on page 66.

  – If the data type of *expression-2* is an integer:

    The first argument must be 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-1* is a character or graphic string, it must not be a CLOB or DBCLOB with an actual length that is not greater than 255 bytes, and its value must be one of the following:

91. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

- 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 66.
- 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 275.

expression-2 must be an integer constant in the range 0 to 12 representing the timestamp precision.

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 and the data type of *expression-2* is not an integer:

  The result is a TIMESTAMP(6) with the date specified by the first argument and the time specified by the second argument. The fractional seconds part of the timestamp is zero.
- If both arguments are specified and the data type of *expression-2* is an integer:

  The result is a timestamp with the precision specified by the second argument.
- If only one argument is specified and it is a TIMESTAMP(*p*):

  The result is that TIMESTAMP(*p*).
- If only one argument is specified and it is a DATE:

  The result is TIMESTAMP(0).
- If only one argument is specified and it is a string:

  The result is the TIMESTAMP(6) value represented by that string extended as described earlier with any missing time information. If the argument is a string of length 14, the TIMESTAMP has a fractional seconds part of zero.

When a string representation of a timestamp 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 timestamp value.

When a string representation of a 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: If only one argument is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

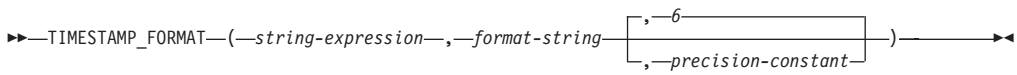
Example

- 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 a built-in character or graphic string that is not a CLOB or DBCLOB, with a length attribute that is not greater than 254. [92] The *string-expression* must contain the components of a timestamp that correspond to the format specified by *format-string*.

format-string
An expression that returns a value of a built-in character or graphic string that is not a CLOB or DBCLOB, with a length attribute that is not greater than 254. [92] *format-string* contains a template of how *string-expression* is interpreted and converted to a timestamp value.

A valid *format-string* must contain at least one format element, must not contain multiple specifications for any component of a timestamp, and can contain any combination of the format elements, unless otherwise noted in Table 32. 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 32. Format elements for the TIMESTAMP_FORMAT function

Format	Unit
DD	Day of month (01-31).

92. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

Table 32. Format elements for the TIMESTAMP_FORMAT function (continued)

Format	Unit
FF or FFn	Fractional seconds. 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.
HH24	Hour of the day (00-24).
MI	Minute (00-59).
MM	Month (01-12).
NNNNNN	Microseconds (000000-999999). Same as FF6.
RR	Last two digits of the adjusted year (00-99).
SS	Seconds (00-59).
YY	Last two digits of the year (00-99).
YYYY	Year (0000-9999).

The RR format element can be used to alter how a specification for a year is to be interpreted by adjusting the value to produce a 4-digit value depending on the last 2 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 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

The following defaults will be used when a format string does not include a format element for one of the components of a timestamp:

year current year, as 4 digits

month current month, as 2 digits

day 01

hour 00

minute
00

second
00

fractional seconds
a number of zeros matching the timestamp precision of the result

Leading zeroes can be specified for any component of the 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 timestamp (such as year, month, day, hour, minutes, seconds) can include less than the maximum number of digits for that component of the 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'.

Examples of valid format strings are:

```
'YYYY-MM-DD'
'YYYY-MM-DD-HH24-MI-SS'
'YYYY-MM-DD-HH24-MI-SS-NNNNNN'
```

precision-constant
An integer constant that specifies the timestamp precision of the result. The value must be in the range 0 to 12. If *precision-constant* is not specified, the timestamp precision defaults to 6.

The result is a timestamp with a precision that is based on *precision-constant*. If either of the first two arguments can be null, the result can be null; if either of the first two arguments is null, the result is the null value.

Note

Syntax alternatives: TO_DATE is a synonym for TIMESTAMP_FORMAT.

Example

- 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')
```

TIMESTAMP_ISO

Returns a timestamp value based on its argument. If the argument is a date or string representation of a date, it inserts zero for the time and microseconds parts 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, character string, or a graphic string.[93]

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 66.

| If expression is a timestamp, the result of the function is a timestamp with the
| same precision as expression. Otherwise, the result of the function is
G TIMESTAMP(6). In DB2 for LUW, the result of the function is always
G TIMESTAMP(6).

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 for maximal portability. For more information, see "CAST specification" on page 149.

Example

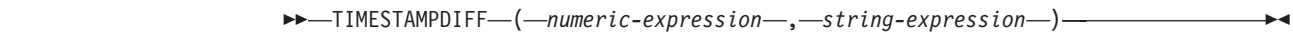
- Assume the following date value:

```
SELECT TIMESTAMP_ISO( DATE( '1988-12-25' ) )
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value '1988-12-25-00.00.00.000000'.

93. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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 are:

1	Fractions of a second
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

string-expression
string-expression must be the equivalent of subtracting two timestamps and converting the result to a string of length 22. If the *string-expression* has more than 6 digits to the right of the decimal point, the string will be truncated to 6 digits. The argument must be an expression that returns a value of a built-in character CHAR or GRAPHIC data type. [94]

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 33. 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

94. In DB2 for LUW, a GRAPHIC data type is only allowed in a Unicode database.

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 34. 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

- Estimate the age of employees 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 a built-in DECFLOAT data type.

expression-2
An expression that returns a value of a built-in DECFLOAT data type.

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
```

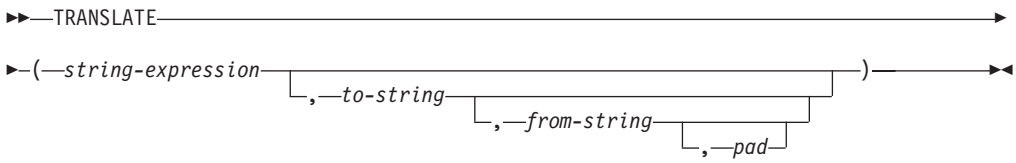
The result of the function is SMALLINT. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

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 *string-expression* may have been converted into other characters.

string-expression
An expression that specifies the string to be converted. *string-expression* must be an expression that returns a value that is a built-in character-string data type. A character string argument must not be a CLOB.

to-string
A string that specifies the characters to which certain characters in *string-expression* are to be converted. This string is sometimes called the *output translation table*.

The string must be a character string constant. A character string argument must not have an actual length that is 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 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 *string-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 *string-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 a character string constant. A character string argument must not have an actual length that is greater than 256.

If *from-string* contains duplicate characters, the left-most one is used and no warning is issued. The default value for *from-string* is a string of all bit representations starting with X'00' and ending with 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 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. For more information, see “UPPER” on page 402.

If more than one argument is specified, the result string is built character by character from *string-expression*, converting characters in *from-string* to the corresponding character in *to-string*. For each character in *string-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.

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.

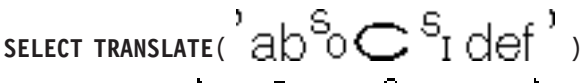
Examples

- Monocase the string 'abcdef'.

```
SELECT TRANSLATE('abcdef')
   FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'ABCDEF'.

- In an EBCDIC environment, monocase the mixed data character string.



  Returns the value 

- 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
```

- Given the same host variable SITE with a value of 'Pivabiska Lake Place':

```
SELECT TRANSLATE(:SITE, '$$', 'Ll')
   FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'Pivabiska $ake P$ace'.

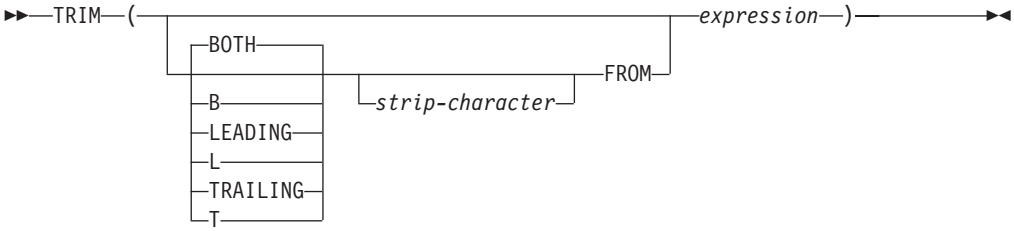
- Given the same host variable SITE with a value of 'Pivabiska Lake Place':

```
SELECT TRANSLATE(:SITE, 'pLA', 'Place', '.')
   FROM SYSIBM.SYSDUMMY1
```

  Returns the value 'pivAbiskA LAk. pLA..'.

TRIM

The TRIM function removes bytes from the beginning, from the end, or from both the beginning and end 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 SBCS or DBCS character that is to be removed. If *expression* is a DBCS graphic 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 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 (X'0020').
- If *expression* is a UTF-8 character string, then the default strip character is a UTF-8 blank (X'20').
- Otherwise, the default strip character is an SBCS blank.

expression
An expression that returns a value of any built-in character string data type, graphic data type, or numeric data type. *expression* must not be a CLOB or DBCLOB. 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 404.

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
GRAPHIC or VARGRAPHIC	VARGRAPHIC

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.

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
```

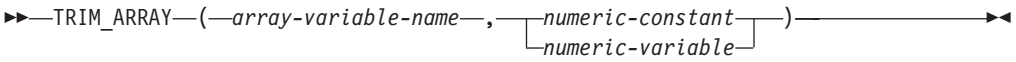
  Results in: '345.50'

- In an EBCDIC environment, assume the string to be stripped contains mixed data.

  SELECT TRIM(BOTH 'SO SI' **FROM** 'SO AB C SI' **)**
  FROM SYSIBM.SYSDUMMY1

  Results in: 'SOAB CSI'

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 an array type.

numeric-constant **or** *numeric-variable*
Specifies the number of elements that will be trimmed from the copy of *array-variable-name*. This must be a constant or SQL variable or parameter that can be cast to the integer data type. The value of must be between 0 and the cardinality of *array-variable-name*.

The result array type is identical to the array type of the first argument but with the cardinality decreased by the number of elements trimmed.

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 or SQL function as the only expression on the right side of an *assignment-statement*.

Example

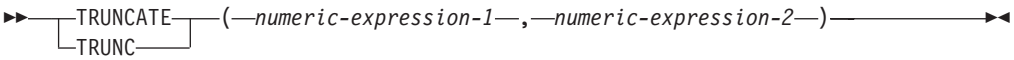
Assume that array type PHONENUMBERS and array variable RECENT_CALLS are defined as follows:

```
CREATE TYPE PHONENUMBERS AS INTEGER ARRAY[50];
DECLARE RECENT_CALLS PHONENUMBERS;
```

The following statement removes the last element from the array variable RECENT_CALLS.

```
SET RECENT_CALLS = TRIM_ARRAY(RECENT_CALLS, 1)
```

TRUNCATE or TRUNC



The TRUNCATE function returns *numeric-expression–1* truncated to *numeric-expression–2* places to the right or left of the decimal point.

numeric-expression–1
An expression that returns a value of any built-in numeric data type.

If *numeric-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.

numeric-expression–2
An expression returns a value that is a small or large integer. The absolute value of integer specifies the number of places to the right of the decimal point for the result if *numeric-expression–2* is not negative, or to the left of the decimal point if *numeric-expression–2* is negative.

If *numeric-expression–2* is not negative, *numeric-expression–1* is truncated to the *numeric-expression–2* number of places to the right of the decimal point.

If *numeric-expression–2* is negative, *numeric-expression–1* is truncated to 1 + (the absolute value of *numeric-expression–2*) number of places to the left of the decimal point. If 1 + (the absolute value of *numeric-expression–2*) is greater than or equal to the number of digits to the left of the decimal point, the result is 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),
       TRUNCATE(873.726, -4)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns (leading zeroes are shown to demonstrate the precision and scale of the result):

```
0873.720   0873.700  0873.000  0870.000  0800.000  0000.000  0000.000
```

- Calculate both positive and negative numbers.

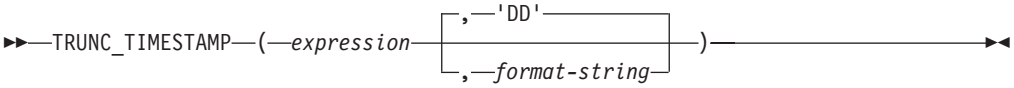
```
SELECT TRUNCATE( 3.5, 0),
       TRUNCATE( 3.1, 0),
       TRUNCATE(-3.1, 0),
       TRUNCATE(-3.5, 0)
  FROM TABLEX
```

This example returns:

```
3.0   3.0  -3.0  -3.0
```

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. [95]

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 with an actual length that is not greater than 255 bytes. It is first converted to a TIMESTAMP(12) value. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 66.

format-string
An expression that returns a built-in character string data type or graphic string data type that is not a CLOB of DBCLOB. The actual length of the expression must not be greater than 254 bytes.[95] *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 not removed from the string. The resulting substring must be a valid format element for a timestamp.

Allowable values for *format-string* are listed in Table 31 on page 354.

The result of the function is a timestamp with a timestamp precision of:

- *p* when the data type of expression is TIMESTAMP(*p*)
- 6 otherwise.

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 truncated 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.

95. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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 402.

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 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.

string-expression
An expression that specifies the string to be converted. *String-expression* must return a value that is a built-in character or graphic data type that is not a CLOB or DBCLOB.

G In DB2 for LUW, a graphic string is only allowed in a Unicode database. If a
G supplied argument is a graphic string, it is first converted to a character string
G before the function is executed.

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 can be specified in place of UPPER.

Examples

- Make the string 'abcdef' uppercase using the UPPER scalar function.

```
SELECT UPPER('abcdef')
     FROM SYSIBM.SYSDUMMY1
```

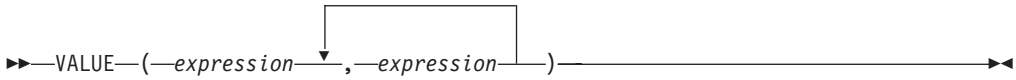
  Returns the value 'ABCDEF'.

- In EBCDIC environments, make the mixed data character string uppercase using the UPPER scalar function.

  UPPER('ab<SO>c<SI>def')

  Returns the value: 'AB<SO>C<SI>DEF'

VALUE



The VALUE function returns the value of the first non-null expression.

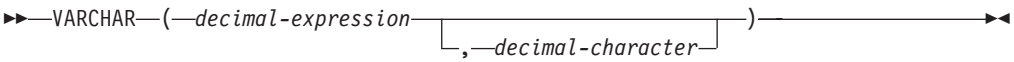
Notes

Syntax alternatives: The VALUE function can be specified in place of the COALESCE function. COALESCE should be used for conformance to SQL 2011 Core. For more information, see "COALESCE" on page 236.

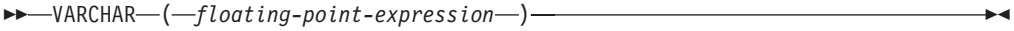
VARCHAR

Integer to Varchar

Decimal to Varchar

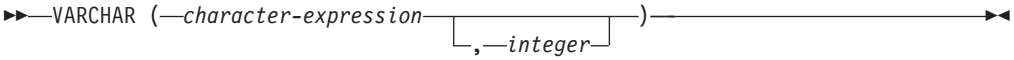


Floating-point to Varchar

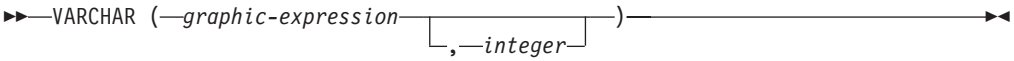


Decimal floating-point to Character

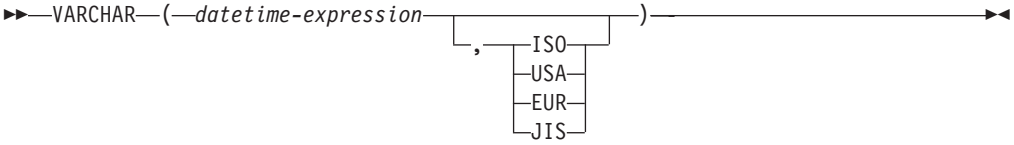
Character to Varchar



Graphic to Varchar



Datetime to Varchar



The VARCHAR function returns a varying-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 a Unicode 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 first argument must be a built-in data type other than a BLOB.

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 a built-in SMALLINT, INTEGER, or BIGINT data type.

The result is the varying-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 attribute of the result is 6.
- If the argument is a large integer, the length attribute is 11.
- If the argument is a big integer the length attribute 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.

The CCSID of the string is the default SBCS CCSID at the current server.

Decimal to Varchar

decimal-expression
An expression that returns a value that is a built-in DECIMAL or NUMERIC data type. If a different precision and scale is desired, 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 100.

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. If the scale of decimal-expression is zero, the decimal character is not 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. 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 string 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 built-in floating-point data type (DOUBLE or REAL).

The single-byte character constant used to delimit the decimal digits in *character-expression* from the whole part of the number is the default decimal point. For more information, see "Decimal point" on page 100.

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 default decimal point 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 default decimal point. If the argument is zero, the result is 0E0.

The CCSID of the string 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 built-in decimal floating-point data type (DECFLOAT).

The single-byte character constant used to delimit the decimal digits in *character-expression* from the whole part of the number is the default decimal point. For more information, see "Decimal point" on page 100.

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 default decimal point. 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 default decimal point. 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 string 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 character-string data type. The argument must not be a CLOB and must have an actual length that is not greater than 32672 bytes.

integer
An integer constant that specifies the length attribute for the resulting varying

length character string. The value must be between 1 and 32672. In EBCDIC environments, 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 an empty string constant, the length attribute of the result is 1. In DB2 for LUW the length attribute is 0.
- 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.

The CCSID of the result is the same as the CCSID of *character-expression*.

Graphic to Varchar

graphic-expression
An expression that returns a value that is a built-in Unicode graphic-string data type. The argument must not be a DBCLOB and must have an actual length that is not greater than 16336 characters.

integer
An integer constant that specifies the length attribute for the resulting varying length character string. The value must be between 1 and 32672. In EBCDIC environments, if the result is mixed data, the second argument cannot be less than 4.

If the second argument is not 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. In DB2 for LUW the length attribute is 0.
- 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 product-specific.

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.

The CCSID of the string is the default CCSID at the current server.

Datetime to Varchar

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. The length attribute and actual length of the result is 10. For more information see "String representations of datetime values" on page 66.

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 from 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 66.

TIMESTAMP
The second argument is not applicable and must not be specified.

The result is the character-string representation of the timestamp. If *datetime-expression* is a TIMESTAMP(0), the length attribute and actual length of the result is 19. If the data type of *datetime-expression* is a TIMESTAMP(*n*), where *n* is greater than 0, the length attribute and actual length of the result is 20+*n*. Otherwise 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 66.

The CCSID of the string is the default SBCS CCSID at the current server.

Note

Syntax alternatives: If the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

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 a timestamp in the format indicated by *format-string*.

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 DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00).

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 or timestamp with an actual length that is not greater than 255 bytes. It is first converted to a TIMESTAMP(12) value. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 66.

format-string
An expression that returns a built-in character or graphic string data type that is not a CLOB or DBCLOB, with a length attribute that is not greater than 254 bytes. If the argument is a graphic string representation of a timestamp, it is first converted to a character string before evaluating the function. *format-string* contains a template of how *expression* is to be formatted. The resulting value may contain characters in any case.

A valid *format-string* can contain a combination of the format elements. 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 ( )

Table 35. Format elements for the VARCHAR_FORMAT function

Format	Unit
AM or PM	Meridian indicator (morning or evening) without periods. The meridian indicator depends on the national language (locale). DB2 for z/OS uses the exact strings.
A.M. or P.M.	Meridian indicator (morning or evening) with periods. This format element uses the exact strings 'A.M.' or 'P.M.'.
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.
D	Day of week (1-7).

96. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

Table 35. Format elements for the VARCHAR_FORMAT function (continued)

Format	Unit
DD	Day of month (01-31).
DDD	Day of year (001-366).
FF or FF*n*	Fractional seconds (000000-999999). The number *n* is used to specify the number of digits to include in the value returned. Valid values for n are 1-6. Specifying FF is equivalent to specifying FF6.
HH	Hour of the day (01-12).
HH12	Hour of the day (01-12).
HH24	Hour of the day (00-24).
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	Name of the month in uppercase, titlecase, or lowercase format. The name of the month depends on the national language (locale). DB2 for z/OS uses English.
MON, Mon, or mon	Abbreviated name of the month in uppercase, titlecase, or lowercase format. The name of the month depends on the national language (locale). DB2 for z/OS uses English.
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).
YY	Last two digits of the year (00-99).
YYY	Last three digits of the year (000-999).
YYYY	Year (0000-9999).

Note: The format elements in Table 1 are not case sensitive, except for the following:

- AM, PM
- A.M., P.M.
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

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 *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, 'DDDDD' is interpreted as the format elements, 'DDD' and 'DD'.

The result is the varying-length character string that contains *expression* in the format specified by *format-string*. *format-string* also determines the length attribute and actual length of the result. The actual length must not be greater than the length attribute of the result. 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 default SBCS CCSID of the current server.

Note

Syntax alternatives: TO_CHAR is a synonym for VARCHAR_FORMAT.

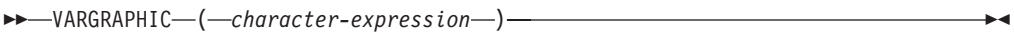
Example

- Set the character variable TVAR to the timestamp value of RECEIVED from CORPDATA.IN_TRAY, using the character string format supported by the function to specify the format of the value for TVAR.

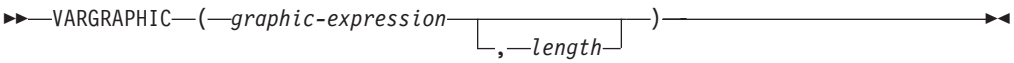
```
SELECT VARCHAR_FORMAT(RECEIVED,'YYYY-MM-DD HH24:MI:SS')
  INTO :TVAR
  FROM CORPDATA.IN_TRAY
```

VARGRAPHIC

Character to Vargraphic



Graphic to Vargraphic



The VARGRAPHIC function returns a varying-length graphic-string representation of:

- 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 expression is an empty string or the EBCDIC string `X'0E0F'`, the result is an empty string.

Character to Vargraphic

character-expression
An expression that returns a value that is a built-in character-string data type. It cannot be bit data.[97]

If the expression is an empty string or the EBCDIC string `X'0E0F'`, the length attribute of the result is 1. In DB2 for LUW the length attribute of an empty string is 0. 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 *character-expression*, as measured in single-byte characters, is greater than the specified length of the result, as measured in double-byte characters, the result is truncated.

The CCSID of the result is determined by a mixed data CCSID. For more information, seeDetermining the Graphic Result CCSID.

Graphic to Vargraphic

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 of the result and must be an integer constant between 1 and 16336.

If the second argument is not specified,

- If the expression is an empty string, the length attribute of the result is 1. In DB2 for LUW the length attribute is 0.

97. Although DB2 for z/OS supports storing ASCII data, expression can only be in the EBCDIC encoding scheme.

- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result depends on the number of characters in *graphic-expression*. If the length of *graphic-expression* is greater than the length specified, the result is truncated.

The CCSID of the result is the CCSID of *graphic-expression*.

Note

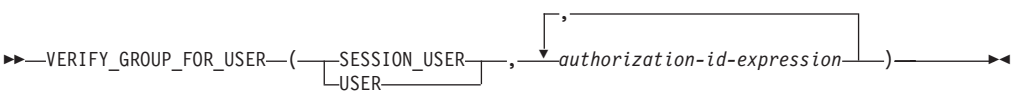
Syntax alternatives: If the first argument is *graphic-expression* and the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 149.

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'
```

VERIFY_GROUP_FOR_USER



The VERIFY_GROUP_FOR_USER function returns a value that indicates whether the run-time authorization ID is in the list of authorization IDs or is a member of any of the group authorization IDs specified by the list of *authorization-id-expression* arguments.

SESSION_USER or USER
Specifies the run-time authorization.

authorization-id-expression
An expression that specifies an authorization name. The existence of the authorization name at the current server is not verified. *authorization-id-expression* must return a value of any built-in character-string or graphic-string data type that is not a LOB. The value of each *authorization-id-expression* must have a length of at least 1 and must not exceed the length of a authorization name. See Chapter 9, "SQL limits," on page 959. The content of the string is not folded to uppercase and is not left justified.

The result of the function is a large integer. The result cannot be null.

The value of the result is 1 if the authorization ID represented by the first argument or the groups that authorization ID is a member of is anywhere in the list of *authorization-id-expression*s. Otherwise, the result is 0.

The VERIFY_GROUP_FOR_USER function is deterministic within a connection. It is not deterministic across connections. It can be referenced in a CREATE MASK or CREATE PERMISSION statement to verify access to the data.

Example

- Assume that table EMPLOYEE exists and that column level access control is activated for the table. Alex (who has security administrator authority) created the following column mask to control what information is returned for a social security number depending on who requests the information. The column mask only returns the actual social security number if the session user is a member of the MGR group profile. Otherwise a masked representation of the social security number is returned.

```
CREATE MASK SSN_MASK
       ON EMPLOYEE
       FOR COLUMN SSN
   RETURN
     CASE
       WHEN VERIFY_GROUP_FOR_USER(SESSION_USER,'MGR') = 1
            THEN SSN
       ELSE 'XXX-XX-' CONCAT SUBSTR(SSN, 8,4)
     END
   ENABLE;
```

  An ALTER TABLE statement is then issued to activate the column mask on the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE
     ACTIVATE COLUMN ACCESS CONTROL;
COMMIT;
```

- Assume that Mary is a manager and is a member of the MGR group profile. Mary issues the following statement:

  ```
  SELECT SSN FROM EMPLOYEE WHERE NAME = 'Tom';
  ```

  The SSN_MASK column mask is applied to the SSN column to produce the result table. Since Mary is a member of the MGR group profile, the result table contains Tom's actual social security number.

  Later, Mary is no longer a manager and is removed from the MGR group profile. She issues the same query again:

  ```
  SELECT SSN FROM EMPLOYEE WHERE NAME = 'Tom';
  ```

  As before, the SSN_MASK column mask is applied to the SSN column to produce the result table. This time the result table contains a masked version of Tom's social security number where only the last 4 digits of the actual number are returned. An 'X' is returned for the other digits.

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.[98]

If *expression* is a string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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 ('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.

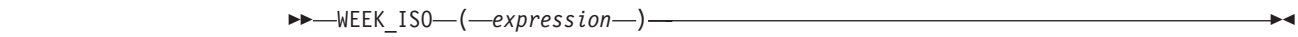
```
SELECT DATE_1, WEEK(DATE_1)
  FROM X
```

  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
2000-12-31    54
```

98. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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.[99]

If *expression* is a string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.

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
2000-12-31    52
```

99. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

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.[100]

- If *expression* is a string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 66.
- 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 137.

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 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)
```

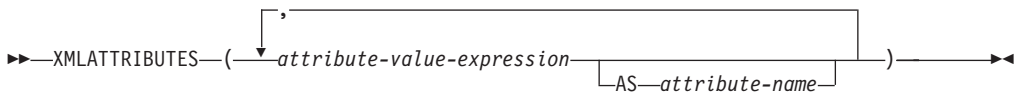
- Select all the projects in the PROJECT table that are scheduled to take less than one year to complete.

```
SELECT *
  FROM PROJECT
  WHERE YEAR(PRENDATE - PRSTDATE) < 1
```

100. In DB2 for LUW, a graphic string is only allowed in a Unicode database.

XMLATTRIBUTES

The XMLATTRIBUTES function constructs XML attributes from the arguments.



The function name cannot be specified as a qualified name.

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 a CLOB, DBCLOB, BLOB, XML, or a distinct type that is based on a CLOB, DBCLOB, or BLOB. 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.

The function name cannot be specified as a qualified name.

string-expression
An expression that returns a value of any built-in character-string that is not a CLOB. 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'.[101]

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
```

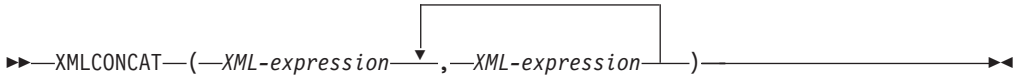
This query produces the following result:

```
<!--This is an XML comment-->
```

101. 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.



The function name cannot be specified as a qualified name.

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 CLOB(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.

The function name cannot be specified as a qualified name.

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.

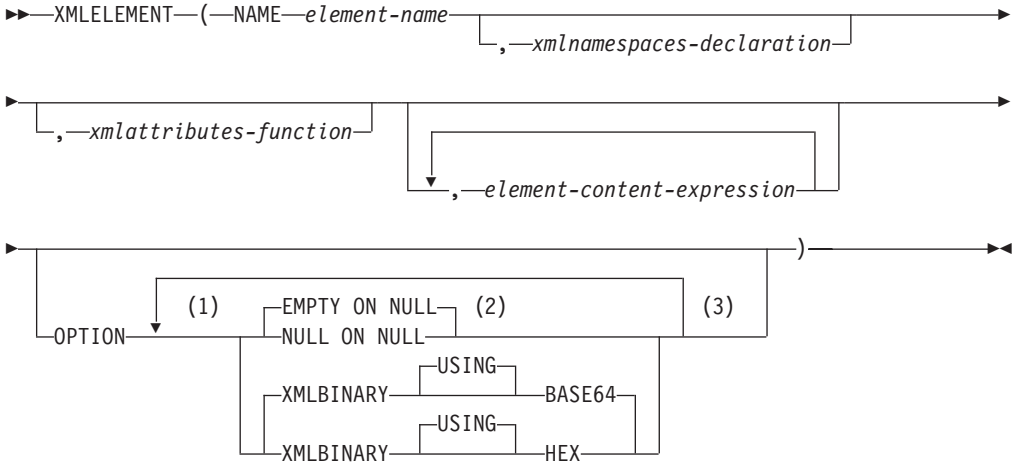
Example

- 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.

The function name cannot be specified as a qualified name.

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 431 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 419 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. 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:

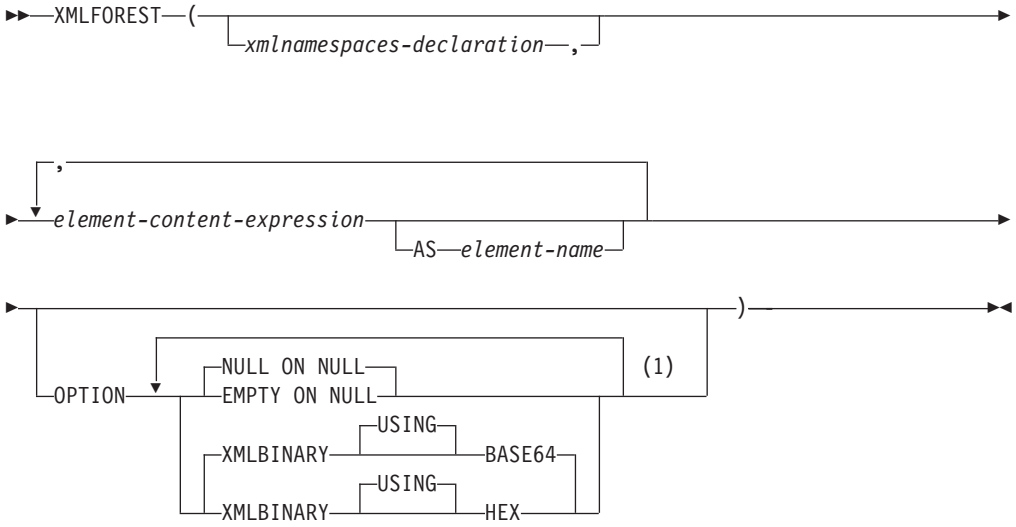
```
SELECT E.EMPNO, E.FIRSTNME, E.LASTNAME,
       XMLELEMENT(NAME "foo:Emp",
                  XMLNAMESPACES('http://www.foo.com' AS "foo"),
                  XMLATTRIBUTES(E.EMPNO AS "serial"),
                  E.FIRSTNME, E.LASTNAME
                  OPTION EMPTY ON NULL) AS "Result"
FROM SESSION.CANDIDATES 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.

The function name cannot be specified as a qualified name.

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 431 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 expression can be any SQL expression of any SQL data type. 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, 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.

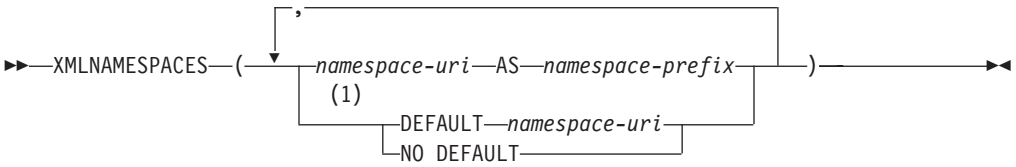
```
SELECT EMPNO,
       XMLFOREST(XMLNAMESPACES(DEFAULT 'http://hr.org',
                               'http://fed.gov' AS "d"),
                 LASTNAME, JOB AS "d:job") AS "Result"
FROM EMPLOYEE WHERE EDLEVEL = 12
```

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.

The function name cannot be specified as a qualified name.

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".

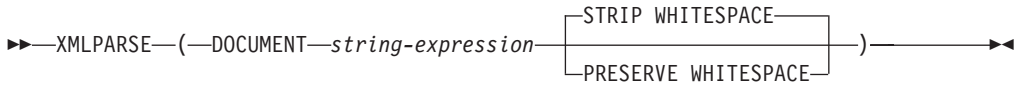
```
SELECT EMPNO,
       XMLSERIALIZE(XMLFOREST(XMLNAMESPACES(DEFAULT 'http://hr.org',
                                            'http://fed.gov' AS "d"),
                              LASTNAME, JOB AS "d:job")
                    AS CLOB(200))
  FROM EMPLOYEE WHERE EDLEVEL = 12
```

  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.



The function name cannot be specified as a qualified name.

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 or BLOB value. 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 up to 1000 bytes will be stripped, unless the nearest containing element has the attribute *xml:space='preserve'*. If any text node begins with more that 1000 bytes of whitespace, an error is returned. 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 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.

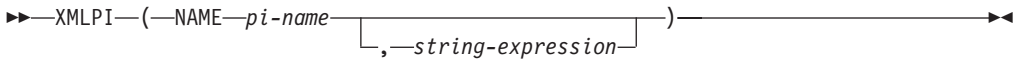
```
INSERT INTO EMP (ID, XVALUE) VALUES(1001,
   XMLPARSE(DOCUMENT
            '<a xml:space=''preserve''> <b xml:space=''default''> <c>c</c>b </b> </a>'
           STRIP WHITESPACE))
```

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.



The function name cannot be specified as a qualified name.

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 of any built-in character-string that is not a CLOB. 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'.[102]

If *string-expression* does not conform to the previous rules, and 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
```

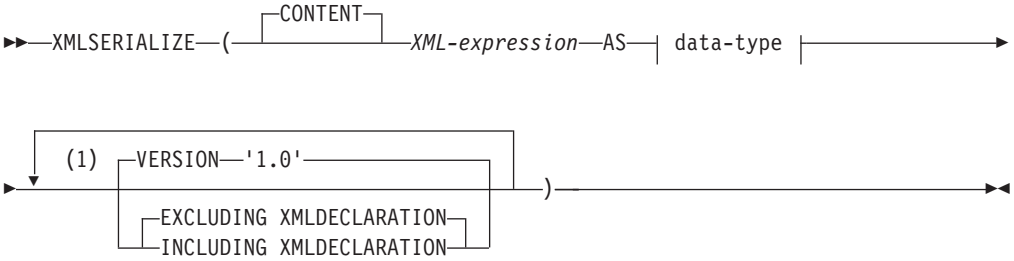
This query produces the following result:

```
<?Warning ?>
```

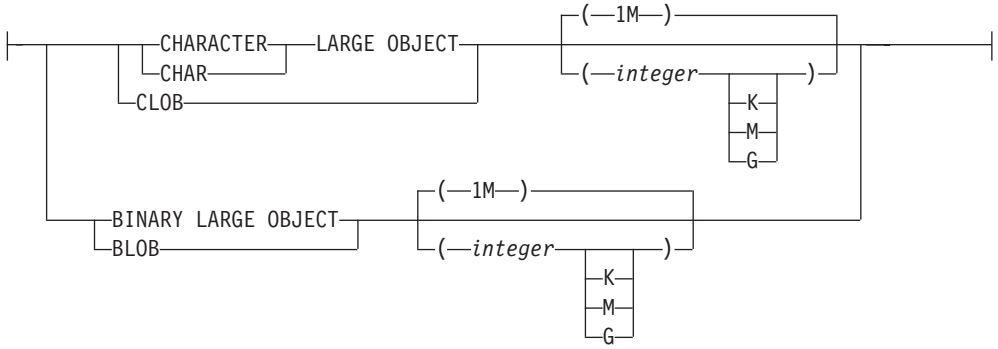
102. Valid Unicode characters consist of the following Unicode code points: #x9, #xA, #xD, #x20-#xD7FF, #xE000-#xFFFD, #x10000-#x10FFFF.

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.

The function name cannot be specified as a qualified name.

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.

G The CCSID of the resulting character string is product specific.

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="UTF-8"?>.

The data type and length attribute of the result are determined from the specified *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, 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) ) 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)
                    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">JOHNSMITH</xmp:Emp>
```

XMLTEXT

The XMLTEXT function returns an XML value that contains the value of *string-expression*.

The function name cannot be specified as a qualified name.

string-expression
An expression that returns a value of any built-in character-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.

```
SELECT XMLTEXT (
         'The stock symbol for Johnson&Johnson is JNJ.')
  FROM SYSIBM.SYSDUMMY1
```

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:

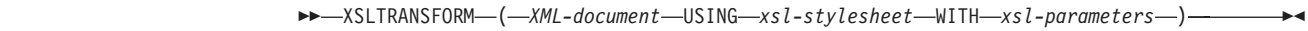
```
SEQNO  PLAINTEXT                                                  EMPHTEXT
-----  ---------------------------------------------------------  ----------
1      This query shows how to construct                          mixed content
2       using XMLAGG and XMLTEXT. Without                         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>
```

XSLTRANSFORM



G DB2 for z/OS does not support the USING and WITH keywords. Instead commas
G must separate the *XML-document*, *xsl-stylesheet*, and *xsl-parameters*.

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.

XML-document
An expression that returns a well-formed XML document with a data type of CHAR, VARCHAR, or CLOB(2 MB). This is the document that is transformed using the XSL style sheet specified in *xsl-stylesheet*.

xsl-stylesheet
An expression that returns a well-formed XML document with a data type of CHAR, VARCHAR, or CLOB(256 KB). The document is an XSL style sheet that conforms to the XSLT Version 1.0 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
An expression that returns a well-formed XML document with a data type of CHAR, VARCHAR, or CLOB(64 KB). 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.

G The result of the function is CLOB(2G). In DB2 for z/OS, the result of the function
G is CLOB(2MB).[103]

If either *XML-document* or *xsl-stylesheet* is null, the result is the null value.

The CCSID of the result is the default CCSID of the current server. Conversion that results in data loss can occur when storing any of the above documents in a character data type.

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.

103. DB2 for z/OS requires IBM® SDK for z/OS®, Java Technology Edition Version 6. DB2 for i requires the XML Toolkit for IBM i and International Components for Unicode (ICU option).

```
CREATE TABLE XML_TAB (c1 INT, xml_doc CLOB(2M), xsl_doc CLOB(256K));
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
  WITH '<params xmlns="http://www.ibm.com/XSLTransformParameters"></params>')
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.

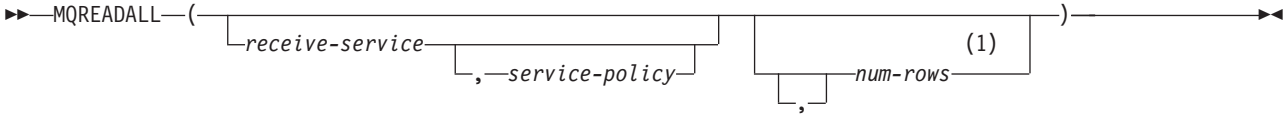
Table functions

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.

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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

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 36. Format of the resulting table for MQREADALL

Column name	Data type	Contains
MSG	VARCHAR(4000) [1]	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	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

Notes:

1. See Table 63 on page 967 for more information.

The CCSID of the result columns is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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.

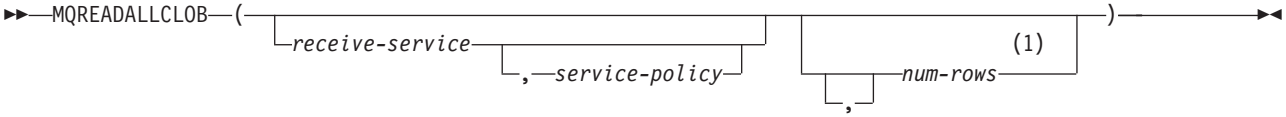
```
SELECT *
 FROM TABLE (MQREADALL ()) AS T
 WHERE T.CORRELID = '1234'
```

- 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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

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 37. Format of the resulting table for MQREADALLCLOB

Column name	Data type	Contains
MSG	CLOB(1M) [1]	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

Notes:

1. See Table 63 on page 967 for more information.

The CCSID of the result columns is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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.

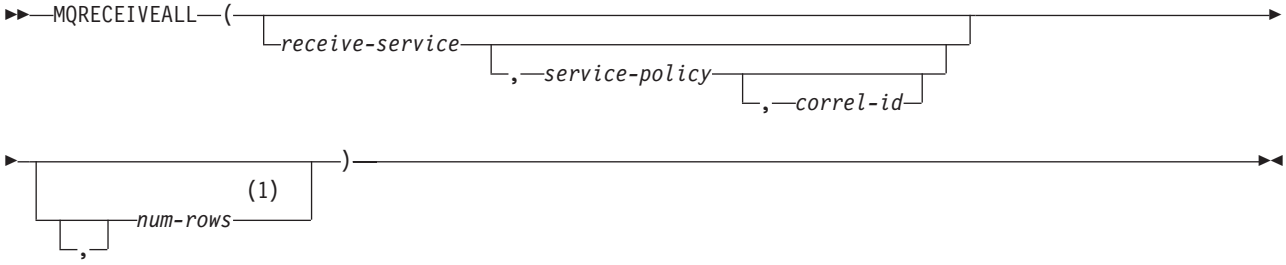
```
SELECT *
 FROM TABLE (MQREADALLCLOB ()) AS T
 WHERE T.CORRELID = '1234'
```

- 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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

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.

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 similar to that shown in the following table. All the columns are nullable.

Table 38. Format of the resulting table for MQRECEIVEALL

Column name	Data type	Contains
MSG	VARCHAR(4000) [1]	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	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

Notes:

1. See Table 63 on page 967 for more information.

The CCSID of the result columns is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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.

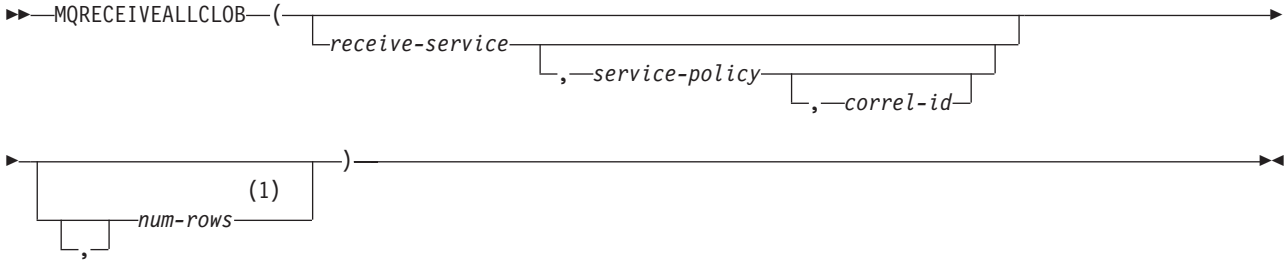
```
SELECT *
 FROM TABLE (MQRECEIVEALL ('MYSERVICE','MYPOLICY','1234')) AS T
```

- 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 MQ service repository. The MQ service repository is product specific. 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.

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 MQ policy repository.The MQ service repository is product specific. 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.

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.

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 39. Format of the resulting table for MQRECEIVEALLCLOB

Column name	Data type	Contains
MSG	CLOB(1M) [1]	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

Notes:

1. See Table 63 on page 967 for more information.

The CCSID of the result columns is the default CCSID at the current server.

Notes

Prerequisites: In order to use the MQSeries functions, IBM MQSeries must be installed, configured, and operational.

Schema: DB2 for i includes the MQ functions in the same schema as other built-in functions. DB2 for LUW and DB2 for z/OS define the MQ functions in product specific schemas that are not included in the default SQL path.

Privileges: Privileges may be granted to or revoked from an MQ function. The EXECUTE privilege is required to use an MQ function.

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.

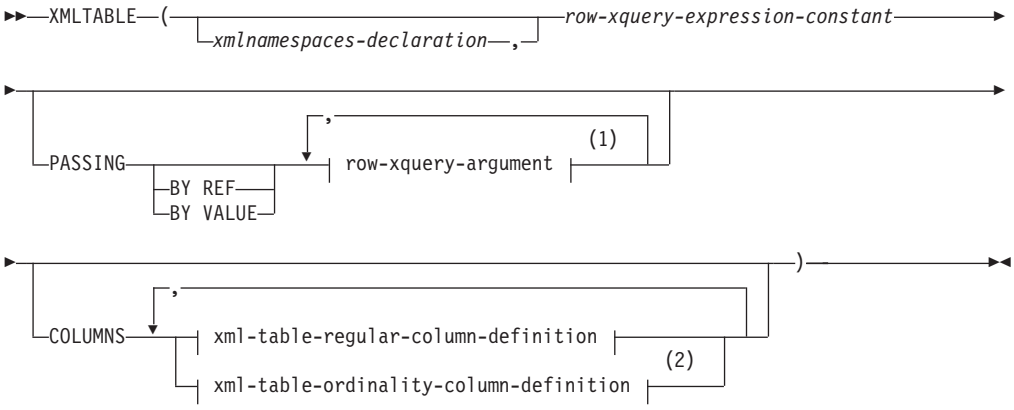
```
SELECT *
 FROM TABLE (MQRECEIVEALLCLOB ('MYSERVICE','MYPOLICY','1234')) AS T
```

- 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.

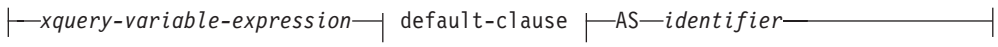
```
SELECT *
 FROM TABLE (MQRECEIVEALLCLOB (10)) AS T
```

XMLTABLE

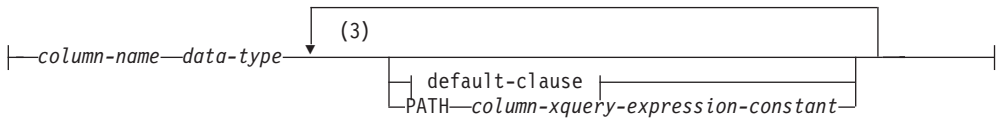
The XMLTABLE function returns a result table from the evaluation of XPath expressions, possibly using specified input arguments as XPath variables. Each item in the result sequence of the row XPath expression represents a row of the result table.



row-xquery-argument:



xml-table-regular-column-definition:



xml-table-ordinality-column-definition:

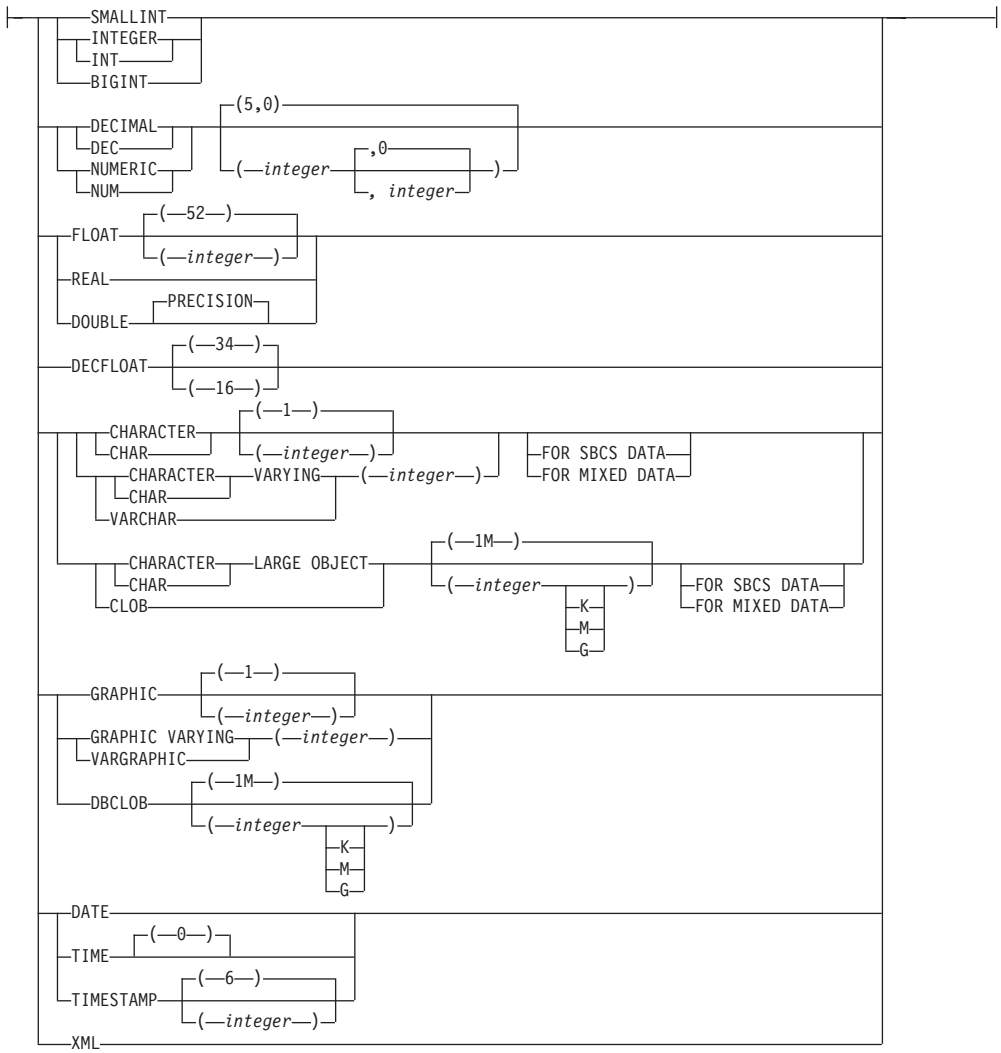
Notes:

1 *xquery-context-item-expression* must not be specified more than one time.

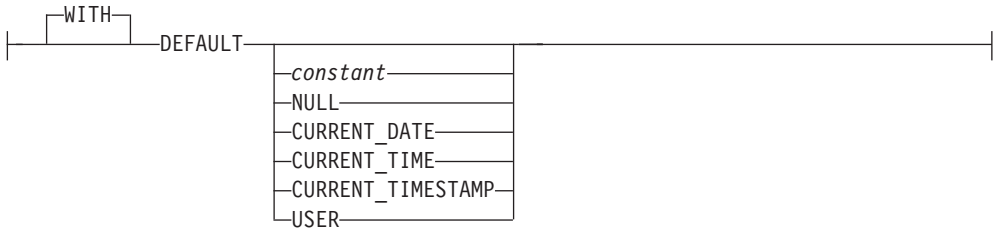
2 The *xml-table-ordinality-column-definition* clause must not be specified more than one time.

3 Neither the *default-clause* nor the PATH clause can be specified more than one time.

data-type:



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 REF
Specifies that any XML input arguments are, passed by reference. When XML values are passed by reference, the XPath evaluation uses the input node trees, if any exist, directly from the specified input expressions and preserves all properties, including the original node identities and document order.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

G DB2 for z/OS and DB2 for LUW support BY REF.

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 clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

G DB2 for i supports BY VALUE.

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. *row-xquery-argument* is an SQL expression that returns a value that is not a LOB, DATE, TIME, TIMESTAMP, REAL, DECFLOAT, or character string with FOR BIT DATA attribute.

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.

Table 40. 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	
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 be 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.

Table 40. Supported SQL to XML conversions (continued)

SQL type	XML type	Notes
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 715. 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, FOR BIT DATA is not allowed.

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 41. Supported XML to SQL result column conversions

XML type	SQL type	Notes
xs:integer	SMALLINT INTEGER BIGINT	
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.

Table 41. Supported XML to SQL result column conversions (continued)

XML type	SQL type	Notes
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 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.
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.

Table 41. Supported XML to SQL result column conversions (continued)

XML type	SQL type	Notes
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 target timestamp type has a precision less than 12, the fractional seconds part of the xs:dateTime value is truncated to the target timestamp 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 6. 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 872.

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:

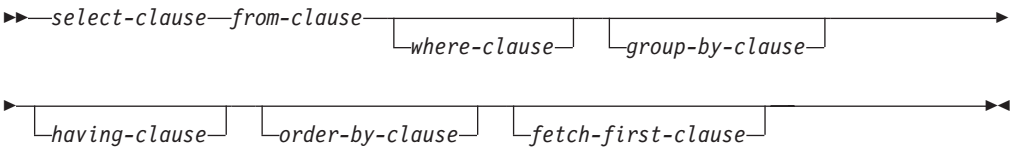
- For each table or view identified in the statement, one of the following:
  - The SELECT privilege on the table or view
  - Ownership of the table or view
- Database administrator authority.

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, one of the following:
  - The EXECUTE privilege on the function
  - Ownership of the function
- Database administrator authority.

If the query references a table that contains active row or column access control and row permissions or column masks are defined for the table, the authorization ID of the statement does not need authority to reference objects that are specified in the definitions of those row permissions or column masks.

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.)

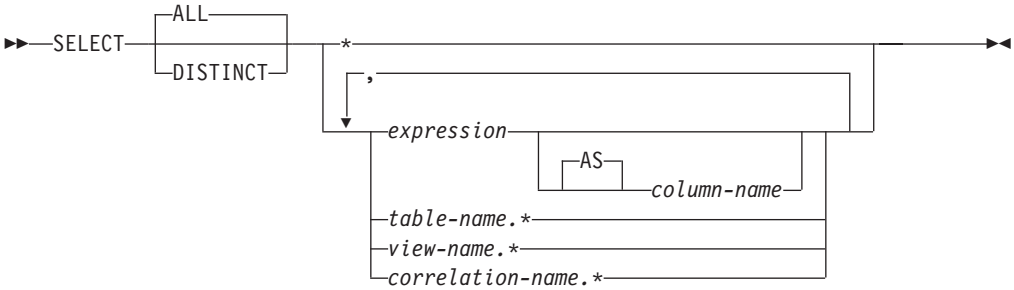
When a subselect directly or indirectly references a table for which row or column access control is enforced, the rules that are defined in the row permissions or column masks affect how the rows in the result table are derived. Typically those rules are based on the authorization ID of the process.

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 logical sequence of the operations is:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause
7. 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
Retains all rows of the final result table. 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.)

When SELECT DISTINCT is specified, no column in the list of column names can be a LOB or XML (or a distinct type that is based on a LOB).

Column access controls do not affect the operation of SELECT DISTINCT. The elimination of duplicated rows is based on the original column values, not the masked values. However, after the application of column masks, the masked values in the final result table might not reflect the uniqueness that is enforced by SELECT DISTINCT.

If a column mask is applied to a column that directly or indirectly derives the result of SELECT DISTINCT, SELECT DISTINCT can return a result that is not deterministic. The following conditions are a few examples of when a result that is not deterministic might be returned:

- The definition of the column mask references other columns of the table to which the column mask is applied.
- The column is referenced in the argument of a built-in scalar function, such as COALESCE, IFNULL, NULLIF, MAX, MIN, LOCATE, etc.
- The column is referenced in the argument of an aggregate function.
- The column is embedded in an expression and the expression contains a function that is not deterministic or has an external action.

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.

* cannot be used in the definition of a row permission or a column mask.

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 will not be 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.

In all the products, SQL statements can be implicitly or explicitly rebound (prepared again). The effect of a rebind on statements that include * or name.* is as follows:

- In DB2 for z/OS and DB2 for LUW, the list of names is reestablished. Therefore, the number of columns returned by the statement may change.
- In DB2 for i, the list of names is normally not reestablished. Therefore, the number of columns returned by the statement will not change. There are cases, however, where the list of names is reestablished. See the product documentation for details.

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 when the statement is prepared). The number of columns in the list must not exceed 750. See Table 63 on page 967 for more information.

Applying the select list

The results of applying the select list to R depend on whether or not GROUP BY or HAVING is used:

If GROUP BY or HAVING is used:

- Each *column-name* in the select list must identify a grouping column, be specified within an aggregate function, or be a correlated reference.
- 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.
- If a column mask is used to mask the values in the final result table, a column for which the column mask is applied must satisfy one of the following conditions:
  - The column must be specified in an aggregate function and the definition of the column mask must not reference the following:
    - A scalar fullselect
    - An aggregate function
  - The column must identify a column-name in the GROUP BY clause and the column must not be referenced in an expression in the GROUP BY clause. In

addition, any columns of the same table as the column for which the column mask is applied and are referenced in the definition of the column mask must be identified with a column-name in the GROUP BY clause. These columns must not be referenced in an expression in the GROUP BY clause.

- A column of a non-base table in the select list must be specified in an aggregate function if a column mask is used to mask the column values in the final result table, and the column of a non-base table maps directly or indirectly to a column name or to an expression in a materialized table expression or view to the table where the column mask is applied.

If neither GROUP BY nor HAVING is used:

- The select must not include any aggregate functions, or each *column-name* must be specified within 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.

Effect of column masks on result columns: When column masks are enabled, they determine the values in the final result table of an outermost select list. When a column mask is enabled for a column, if the column appears in the outermost select list (either implicitly or explicitly), the column mask is applied to the column to produce the values for the final result table. If the column itself does not appear in the outermost select list, but is included in the output (for example, it appears in a materialized table expression or a view), the masked value is included in the result table of the table expression or view so that it can be used in the final result table.

The enabled column masks do not interfere with the operations of other clauses within the statement, such as the WHERE, GROUP BY, HAVING, SELECT DISTINCT, and ORDER BY clauses.

The rows that are returned in the final result table remain the same, except that the values in the result rows might be masked. As such, if a column with masked values also appears in an ORDER BY clause with a sort-key expression, the order is based on the original column values (the masked values in the final result table might not reflect that order). Similarly, the masked values might not reflect the uniqueness enforced by a SELECT DISTINCT. If the masked column is embedded in an expression, the result of the expression might be different because the column mask is applied to the column before the expression is evaluated. For example, a column mask on column SSN can change the result of the function COUNT(DISTINCT SSN) because the DISTINCT operation is performed on the masked values.

When the definition of a column mask is applied to an SQL statement to mask column values in the final result table, the semantics of the column mask might conflict with certain SQL semantics in the statement. In these situations, the combination of the statement and the column mask might return an error.

See “ALTER TABLE” on page 547 for more information about the application of enabled column masks.

Null attributes of result columns

Result columns allow null values if they are derived from:

- Any aggregate function but COUNT or COUNT_BIG
- A column that allows null values
- A scalar function or expression with an operand that allows nulls
- 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 an outer select list
- A *scalar-fullselect*
- A user-defined scalar or table function.

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 SQL variable or SQL parameter (without any functions or operators), the result column name is the unqualified name of that SQL variable or SQL parameter.
- All other result column names 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.
a constant	the same as the data type of the constant.
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.
an expression	see “Expressions” on page 130 for a description of data type attributes.
any function	The data type of the result of the function. For a built-in function, see Chapter 5, “Built-in functions,” on page 187 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.

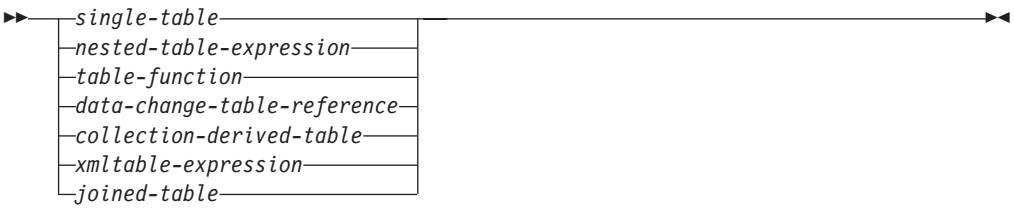
When the expression is:	The data type of the result column is:
the name of any string variable	the same as the data type of the variable, with a length attribute equal to the length of the variable. If the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string.
the name of a datetime column	the same as the data type of the column.
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.

from-clause

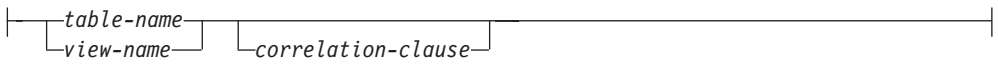
The FROM clause specifies an intermediate result table.

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-reference* (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



single-table:



nested-table-expression:

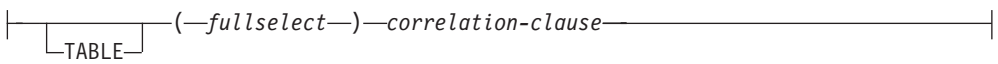
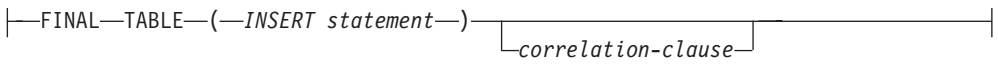
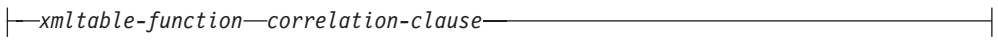


table-function:

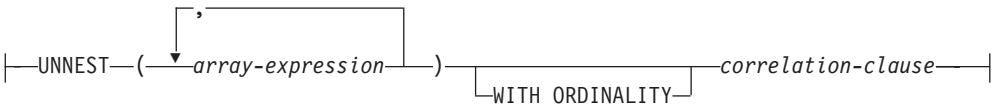
data-change-table-reference:



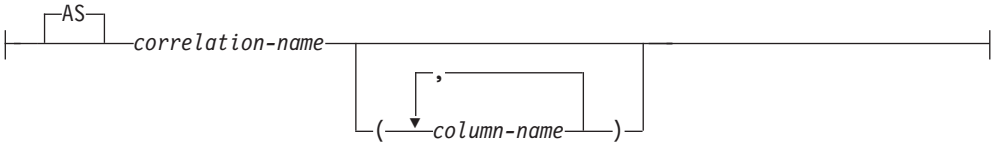
xmltable-expression:



collection-derived-table:



correlation-clause:



A *table-reference* specifies an intermediate result table.

- If a single table or view is identified, the intermediate result table is simply that table or view.
- A fullselect in parentheses is called a *nested table expression*.[104] 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* 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 *xmltable-expression* is specified, the intermediate result table is the set of rows that are returned by the "XMLTABLE" on page 455 function.
-

The list of names in the FROM clause must conform to these rules:

- Each *table-name* and *view-name* must identify 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* 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*
- 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 in "Function resolution" on page 124, 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

104. A nested table expression is also called a *derived 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 *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.

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 must 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*, *collection-derived-table*, or *data-change-table-reference*. 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*, or *data-change-table-reference*. For more information, see "Correlation names" on page 108.

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 desired result table is based on variables.

xmltable-expression: Specifies an invocation of the built-in XMLTABLE table function. See "XMLTABLE" on page 455 for more information.

If a column mask is used to mask the column values in the final result table, and if the result of the XMLTABLE function is used to derive the final result table, the column mask cannot be applied to a column that is specified in the PASSING clause of the XMLTABLE function.

data-change-table-reference: 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* or a SELECT INTO 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.

If row access control is enforced for the target of the data change statement, the rows in the intermediate result table already satisfy the rules that are specified in the enabled row permissions. If column access control is enforced for the target of the data change statement, the enabled column masks are applied to the outermost select list. See "select-clause" on page 465 for more information. If an INCLUDE clause is specified as part of the SQL data change statement, and these additional columns appear in the outermost select list, the column values must not be derived from columns for which column masks are defined.

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.

The content of the intermediate result table for a table reference that contains an SQL data change statement is determined when the cursor is opened. The intermediate result table includes a column for each of the columns of the target table (including implicitly hidden columns) or view. All of the columns of the target table or view of an SQL data change statement are accessible by using the names of the columns from the target table or view unless the columns are renamed by using the correlation clause. If a *correlation-name* is not specified, the column names can by qualified by the target table or view name of the SQL data change statement. If an INCLUDE clause is specified as part of the SQL data change statement, the intermediate result table will contain these additional columns.

collection-derived-table: A collection derived table can be used to unnest the elements of arrays into rows.

array-expression
An expression that returns an array data type. The expression must be one of the following expressions:
- An SQL variable
- An SQL parameter
- A CAST specification of a parameter marker

Names for the result columns produced by the UNNEST function can be provided as part of the correlation-clause of the collection-derived-table clause.

The result table depends on the input arguments.

- If a single array argument is specified, the result is a single column table with a column data type that matches the array element data type.
- 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. The data type of each column matches the data type of the array elements of the corresponding array argument. 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 array index 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 or SQL function.

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 keyword must appear before the fullselect. For more information see "Column name qualifiers to avoid ambiguity" on page 110.

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 is specified. Otherwise, only references to higher levels in the hierarchy of subqueries are allowed.

A nested table expression or table function that contains correlated references to other tables in the same FROM clause:

- Cannot participate in a FULL OUTER JOIN or RIGHT OUTER 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.

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 is in the FROM clause of another nested table expression that contains one of these restrictions.

Examples: 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) GROUP BY E.WORKDEPT)
  AS EMPINFO
  WHERE D.DEPTNO = EMPINFO.DEPT
```

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
  FROM DEPARTMENT D,
  (SELECT AVG(E.SALARY) AS  AVGSAL, COUNT (*) AS EMPCOUNT
    FROM EMPLOYEE E
    WHERE E.WORKDEPT = D.DEPTNO) AS EMPINFO
```

INCORRECT

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
  WHERE t.c3 = z.c4
```

INCORRECT

Example 3: The following example of a table function is valid:

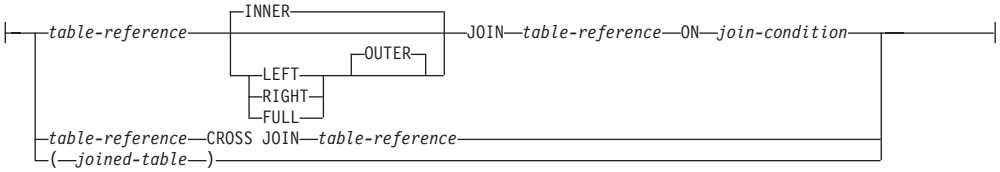
```
SELECT t.c1, z.c5
  FROM t, TABLE(tf4 (2 * t.c2 ) ) AS z
  WHERE t.c3 = z.c4
```

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,
       TABLE(tf7b (a.c6 )) AS b
  WHERE a.c3 = b.c4
```

INCORRECT

joined-table



A *joined-table* specifies an intermediate result table that is the result of either an inner join, outer join, or cross join. The table is derived by applying one of the join operators: INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER, or CROSS to its operands.

If a join operator is not specified, INNER is implicit. The order in which multiple joins are specified 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*. 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
```

Cross joins represent the cross product of the tables, where each row of the left table is combined with every row of the right table. An inner join combines each row of the left table with each row of the right table keeping only the rows where the join-condition is true. Thus, the result table may be missing rows 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 as follows:

- 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 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 any subqueries.
- 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*.
- It cannot include an SQL function.
- Any function referenced in an expression of the join-condition of a full outer join must be deterministic and have no external action.
- For a FULL OUTER JOIN, the predicates can only be combined with AND. In addition, each predicate must have the form 'expression = expression', where one expression references only columns of one of the operand tables of the associated join operator, and the other expression references only columns of the other operand table. The values of the expressions must be comparable.

  Each expression in a FULL OUTER join must include a column name or a cast function that references a column. The COALESCE function is allowed.

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 108 before any rules about which tables the columns must belong to are applied.

Join operations: A *join-condition* 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*. 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* 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 of an operand table. 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* 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.

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.

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 subselect.

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.

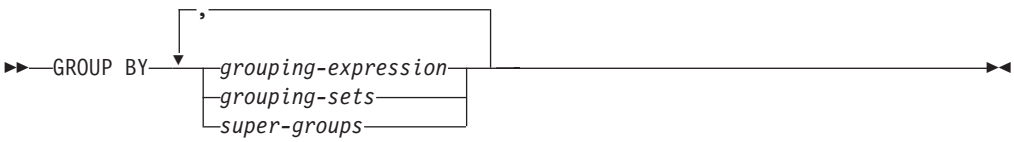
Any subquery in the *search-condition* is effectively executed for each row of R, and the results are used in the application of the *search-condition* to the given row of R. A subquery is executed for each row of R if it includes a correlated reference to a column of R. A subquery with no correlated references is typically executed just once.

If row access controls are enabled for a table and no other row permission is defined, the row access control search condition is the default row permission, 1 = 0. If only one row permission is defined, the row access control search condition is the search conditions that are specified by that permission. Otherwise, if multiple row permissions are defined for a table, the row access control search condition is derived by application of the logical OR operator to the search conditions that are specified by each row permission. This row access control search condition, as a whole, is connected by application of the logical AND operator to the search conditions specified by the WHERE clause and has the same precedence level as other search conditions in the WHERE clause. This process is repeated for each table-reference in the FROM clause of the subselect for which row access controls are enabled.

The row access control search condition acts as a filter to the table-reference to determine the results of the table-reference that are accessible to the authorization ID of the subselect. Because the order in which operators are evaluated is undefined for operators at the same precedence level, other search conditions in the WHERE clause might be evaluated before the row access control search condition. So, the other search conditions have access to the rows that are restricted by the row permission rules. To ensure that sensitive data is protected, the predicates that reference user-defined functions that are defined with the NOT SECURED option are always evaluated after the row access control search condition.

Column access control does not affect the operation of the WHERE clause.

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 LOB data type, an XML data type, or a distinct type that is based on a LOB.
- The length attribute of each *grouping-expression* must not be more than 2000, or 1999 if the expression is nullable
- *grouping-expression* cannot include any of the following items:
  - A correlated column
  - A variable
  - An aggregate function
  - Any function that is non-deterministic or that is defined to have an external action
  - A *scalar-fullselect*
  - A CASE expression whose *search-when-clause* contains a quantified predicate, an IN predicate using a subselect, or an EXISTS predicate.

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 480 and "super-groups" on page 481, respectively.

The result of GROUP BY 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 within a *grouping-expression* are considered equal.

Because every row of a group contains the same value of any *grouping-expression*, a *grouping-expression* can be used in a search condition in a HAVING clause or an expression in a SELECT clause. In each case, the reference specifies only one value for each group.

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.

Column access controls do not affect the operation of the GROUP BY clause. The grouping is performed using the original column values.

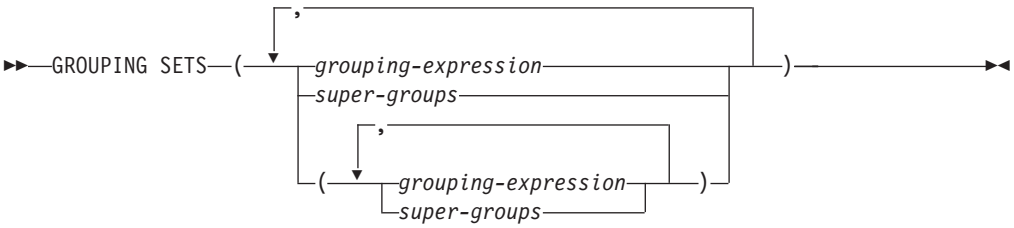
GROUP BY cannot be used in a subquery of a basic predicate or if R is derived from a view whose outer subselect includes GROUP BY or HAVING clauses.

The number of columns must not exceed 120 and the sum of their length attributes must not exceed 16 000. See Table 63 on page 967 for more information.

Row access controls do not affect the operation of the GROUP BY clause.

In certain contexts, the semantics of the column mask can conflict with those in the GROUP BY clause. When this occurs, the column mask cannot be applied for the statement and an error will be returned at bind time. See the "select-clause" on page 465 for more information about how column access controls affect the GROUP BY clause.

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 481.

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
```

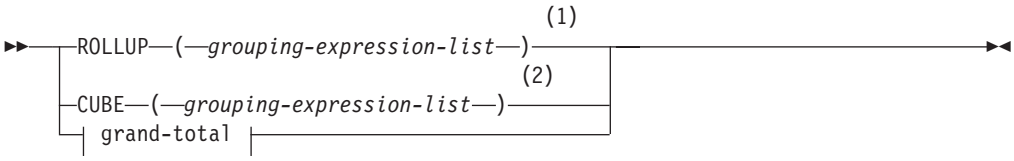
is the same as

```
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.

Example C2 through Example C7 illustrate the use of grouping sets.

super-groups

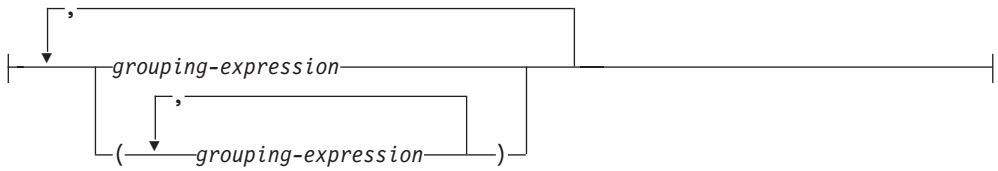


Notes:

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 aggregate 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( C1,C2,...,Cn-1,Cn)
```

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 479. 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 *column-name* in the search condition must do one of the following:

- Unambiguously identify a grouping column 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.

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 "Example 6" on page 498 and "Example 7" on page 499.

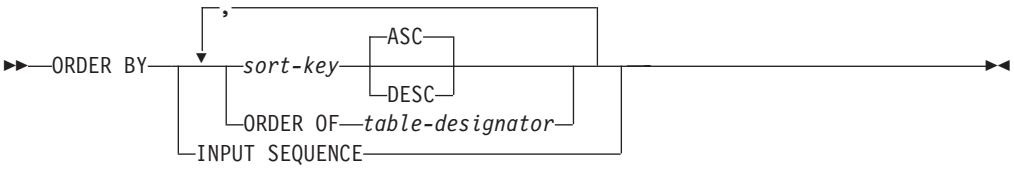
A correlated reference to a group of R must either identify a grouping column or be contained within an aggregate function.

The HAVING clause must not be used in a subquery of a basic predicate or if R is derived from a view whose outer subselect includes GROUP BY or HAVING clauses. When HAVING is used without GROUP BY, any column name in the select list must appear within an aggregate function.

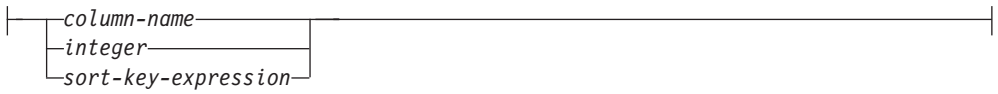
Row access controls do not affect the operation of the HAVING clause.

In certain contexts, the semantics of the column mask can conflict with those in the HAVING clause. When this occurs, the column mask cannot be applied for the statement and an error will be returned at bind time. See the "select-clause" on page 465 for more information about how column access controls affect the HAVING clause.

order-by-clause



sort-key:



The ORDER BY clause specifies an ordering of the rows of the result table.

A subselect that contains an ORDER BY clause cannot be specified in the following objects:

- In the outermost fullselect of a view.
- In the definition of a materialized query table.

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 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 sort 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. A column that is a LOB or XML must not be identified.

A named column in the select list may be identified by a *sort-key* that is an *integer* or a *column-name*. An unnamed column in the select list must be identified by an *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 468 defines when result columns are named, and can be used in the ORDER BY clause. If the *fullselect* includes a UNION operator, see "fullselect" on page 500 for the rules on named columns in a *fullselect*.

Ordering is performed in accordance with the comparison rules described in Chapter 3, "Language elements," on page 43. The null value is higher than all other values. If the ordering specification does not determine a complete ordering, rows with duplicate key values have an arbitrary order. If the ORDER BY clause is not specified, the rows of the result table have an arbitrary order.

Column access controls do not effect the operation of the ORDER BY clause. The order is based on the original column values. However, after column masks are applied, the masked values in the final result table might not reflect the order of the original column values.

The number of *sort-key*s must not exceed 1012 and the sum of their length attributes must not exceed 16 000. See Table 63 on page 967 for more information.

Limits: The use of a *sort-key-expression* or a *column-name* where the column is not in the select list may result in the addition of the column or expression to the temporary table used for sorting. This may result in reaching the limit of the number of columns in a table or the limit on the size of a row in a table. Exceeding these limits will result in an error if a temporary table is required to perform the sorting operation.

column-name
Must unambiguously identify a column of the result table. The column must not be a LOB 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 110 for more information.

If the query is a fullselect that contains a set operator, the *column-name* must not be qualified.

If the query is a subselect, 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 is returned if the subselect:

- specifies DISTINCT in the *select-clause*
- includes aggregate functions in the select list
- includes a GROUP BY clause

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. This column must not be a LOB or XML column.

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*.

A *sort-key-expression* may not reference column names specified in an AS clause in the select list.

The *sort-key-expression* cannot include a non-deterministic function or a function with an external action. The *sort-key-expression* must not be a LOBor XML.

A *sort-key-expression* cannot be specified if DISTINCT is used in the select list of the subselect.

If the query is grouped, the *sort-key-expression* can be an expression in the select list of the query or can include an aggregate function, constant, or variable.

Each expression in the ORDER BY clause must not contain a *scalar-fullselect*.

ASC
Uses the values of the column in ascending order. This is the default.

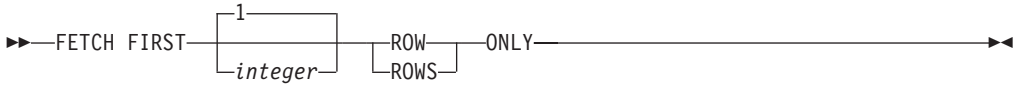
DESC
Uses the values of the column in descending order.

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 subselect (or fullselect) corresponding to the specified *table-designator* must include an ORDER BY clause that is dependant on the data. 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.

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.

fetch-first-clause



The *fetch-first-clause* sets a maximum number of rows that can be retrieved. It lets the database manager know that only *integer* rows should be made available to be retrieved, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond *integer* rows is handled the same way as normal end of data (SQLSTATE 02000). The value of *integer* must be a positive integer (not zero).

A subselect that contains an FETCH FIRST clause cannot be specified in the following objects:

- In the outermost fullselect of a view.
- In the definition of a materialized query table.

Limiting the result table to the first *integer* rows can improve performance. The database manager will cease processing the query once it has determined the first *integer* rows.

If the *order-by-clause* and the *fetch-first-clause* are both specified, the FETCH FIRST operation is always performed on the ordered data. Specification of the *fetch-first-clause* in a *select-statement* makes the result table *read-only*. A *read-only* result table must not be referred to in an UPDATE or DELETE statement. The *fetch-first-clause* cannot appear in a statement containing an UPDATE clause.

Examples of a subselect

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
```

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.

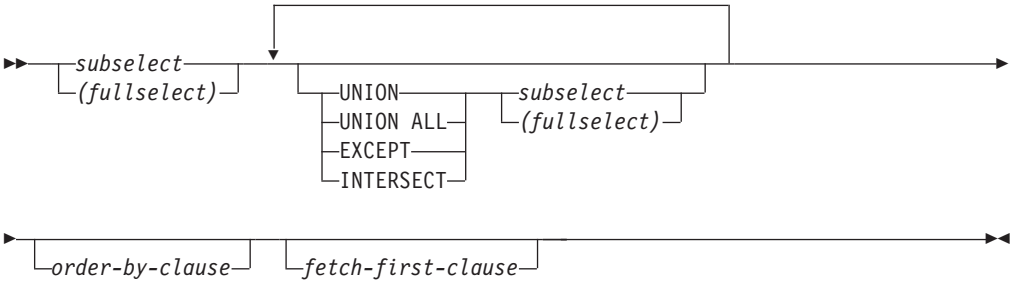
Example 8

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, and UPDATE statement.

A fullselect that is enclosed in parentheses 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.

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 or UNION ALL

If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result consists of all rows in R1 and R2, with the redundant duplicate rows in the result of this operation eliminated. In either case, each row in the result table of the union is either a row from R1 or a row from R2.

Column masks cannot be applied to the select lists that derive the final result table of set operations if any of the set operators that are used to derive the final result table is UNION.

EXCEPT

The result consists of all rows that are only in R1, with the 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.

Column masks cannot be applied to the select lists that derive the final result table of set operations if any of the set operators that are used to derive the final result table is EXCEPT ALL or EXCEPT.

INTERSECT

The result consists of all rows that are in both R1 and R2, with the

duplicate rows in the result of this operation eliminated. Each row in the result table of the intersection is a row that exists in both R1 and R2.

Column masks cannot be applied to the select lists that derive the final result table of set operations if any of the set operators that are used to derive the final result table is INTERSECT ALL or INTERSECT.

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.
- R1 and R2 must not include columns having a data type of CLOB, BLOB, DBCLOB, XML or a distinct type that is based on any of these types. However, this rule is not applicable when UNION ALL is used.
- If the *n*th column of R1 and the *n*th column of R2 have the same result column name, the *n*th column of the result table of the set operation has the same result column name. Otherwise, the *n*th column of the result table of the set operation is unnamed.

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 91.

Duplicate rows: Two rows are duplicates if the value in each column in the first row is equal to the corresponding value of the second row. For determining duplicates, two null values are considered equal.

Operator precedence: When multiple set operations are combined in an expression, set operations within parentheses are performed first. If the order is not specified by parentheses, set operations are performed from left to right with the exception that all INTERSECT operations are performed before any UNION or any EXCEPT operations.

Results of set operators: The following table illustrates the results of all set operations, with rows from result table R1 and R2 as the first two columns and the result of each operation on R1 and R2 under the corresponding column heading.

Table 42. Example of UNION, EXCEPT, and INTERSECT set operations on result tables R1 and R2.

Rows in R1	Rows in R2	Result of UNION ALL	Result of UNION	Result of EXCEPT	Result of 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			

Table 42. Example of UNION, EXCEPT, and INTERSECT set operations on result tables R1 and R2. (continued)

Rows in R1	Rows in R2	Result of UNION ALL	Result of UNION	Result of EXCEPT	Result of INTERSECT
5		3			
		3			
		3			
		3			
		4			
		4			
		4			
		5			

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.

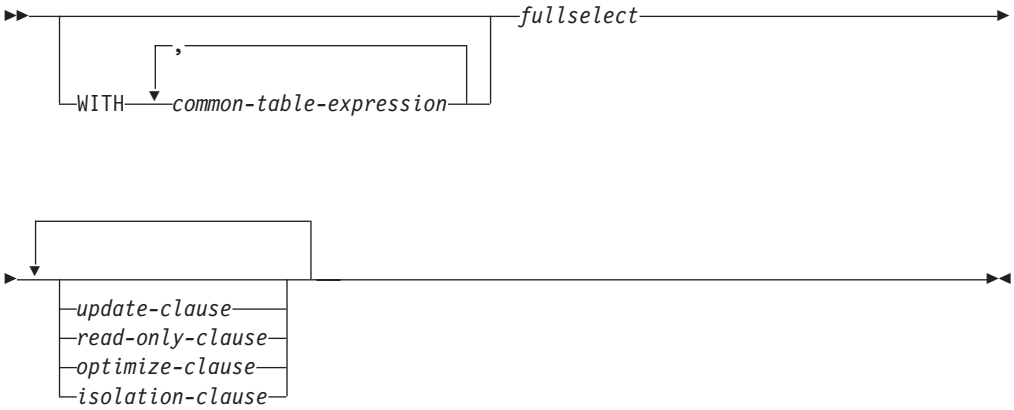
```
(SELECT * FROM T1)
     INTERSECT DISTINCT
(SELECT * FROM T2)
```

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.

select-statement

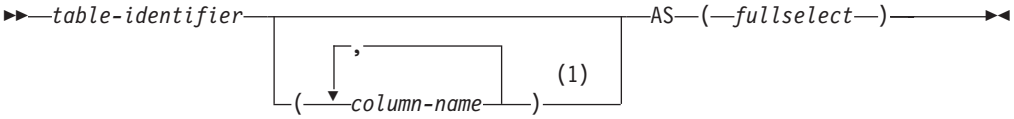


Note:

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.

The *select-statement* is the form of a query that can be directly specified in a DECLARE CURSOR statement 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, using the interactive facility of any of the database managers. In any case, the result table specified by a *select-statement* is the result of the fullselect.

common-table-expression



Notes:

1 If a common table expression is recursive, or if the fullselect results in duplicate column names, column names must be specified.

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.

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 fullselect 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. If the common table expression is specified in an INSERT statement, the *table-identifier* cannot be the same as the table or view name that is the object of the insert. 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 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 updates or deletes are not used)

- To enable grouping by a column that is derived from a *scalar-fullselect* or function that is not deterministic or has external action
- When the desired 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 the *fullselect* of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*. Queries using recursion are useful in supporting applications such as bill of materials, reservation systems, and network planning.

The following 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 any column of 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 is not allowed if the *common-table-expression* is the right operand. RIGHT OUTER JOIN and FULL OUTER JOIN is 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 data type, length, and code page for the column are determined based on the initialization *fullselect*. The corresponding column in the iterative *fullselect* must have the same data type and length as the data type and length determined based on the initialization *fullselect* and the CCSID must match. However, for character string types, the length of the two data types may differ. In this case, the column in the iterative *fullselect* must have a length that would always be assignable to the length determined from the initialization *fullselect*.

When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will terminate. 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.

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 from part '01' we go 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
```

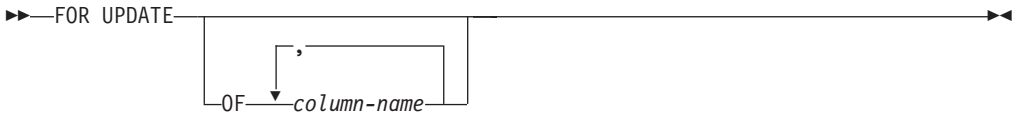
This query is similar to example 1. The column *LEVEL* was introduced to count the levels from the original part. In the initialization fullselect, the value for the *LEVEL* column is initialized to 1. In the subsequent fullselect, the level from the parent is incremented by 1. Then to control the number 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 be updated 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 a dynamically prepared *select-statement* does not contain an UPDATE clause, the cursor associated with that *select-statement* cannot be referenced in a Positioned UPDATE statement.

If a statically prepared *select-statement* does not contain an UPDATE clause 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.

[105]

G If an UPDATE clause is not specified in a statically prepared *select-statement* used
G by an DB2 for i or DB2 for LUW application requester connected to a DB2 for
G z/OS application server, its associated cursor cannot be referenced in a Positioned
G UPDATE statement.

105. In DB2 for z/OS, and DB2 for LUW, a program preparation option must be used if the UPDATE clause is not specified and the cursor is referenced in subsequent Positioned UPDATE statements. If this program preparation option is not used and the UPDATE clause is not specified, the cursor cannot be referenced in a Positioned UPDATE statement. For DB2 for z/OS the program preparation option is STDSQL(YES) or NOFOR, for DB2 for LUW it is LANGLEVEL SQL92E.

read-only-clause

The READ ONLY clause indicates that the result table is read-only.

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.

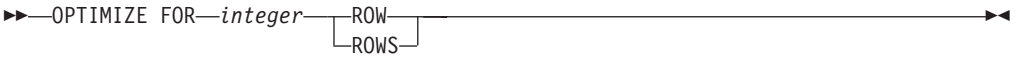
For result tables in which updates and deletes are allowed, specifying FOR READ ONLY can possibly improve the performance of FETCH operations by allowing the database manager to do blocking and avoid exclusive locks. For example, in programs that contain dynamic SQL statements without the READ ONLY or 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

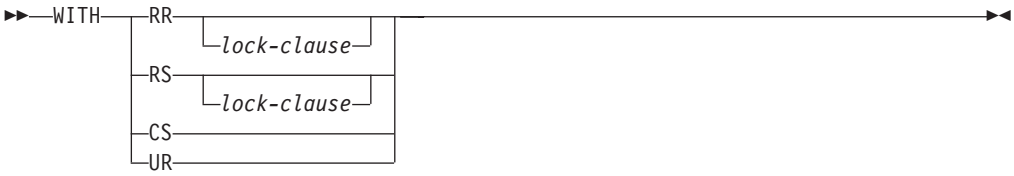


The *optimize-clause* tells the database manager to assume that the program does not intend to retrieve more than *integer* rows from the result table. Without this clause, the database manager assumes that all rows of the result table will be retrieved. Optimizing for *integer* rows can improve performance. The database manager will optimize the query based on the specified number of rows.

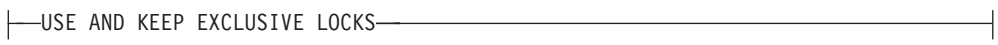
The clause does not change the result table or the order in which the rows are fetched. Any number of rows can be fetched, but performance can possibly degrade after *integer* fetches.

The value of *integer* must be a positive integer (not zero).

isolation-clause



lock-clause:



The optional isolation-clause specifies the isolation level at which the select statement is executed.

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

It is not allowed on updatable cursors.

CS Cursor Stability

UR Uncommitted Read

WITH UR can be specified only if the result table is read-only. If *isolation-clause* is not specified, the default isolation is used with the exception of a default isolation level of uncommitted read. With uncommitted read, the default isolation level of the statement depends on whether the result table is read-only; if the result table is read-only then the default will be UR; if the result table is not read-only then the default will be CS. See “Isolation level” on page 28 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 in all isolation levels other than UR till the end of the unit of work.

Examples of a select-statement

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) AS AVGSAL
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY AVGSAL
```

Example 4

Declare a cursor named UP_CUR to be used in a C program to update 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.

```
EXEC SQL  DECLARE UP_CUR CURSOR FOR
            SELECT PROJNO, PRSTDATE, PRENDATE
              FROM PROJECT
              FOR UPDATE OF PRSTDATE, PRENDATE;
```

Example 5

Select items from a table with an isolation level of Repeatable Read (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
```

Examples of a select-statement

Chapter 7. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements listed in the following tables.

Table 43. SQL Schema Statements

SQL Statement	Description	Reference
ALTER FUNCTION (external)	Alters the description of an external function	"ALTER FUNCTION (external)" on page 530
ALTER FUNCTION (SQL)	Alters the description of an SQL function	"ALTER FUNCTION (SQL)" on page 534
ALTER MASK	Alters the state of a mask	"ALTER MASK" on page 538
ALTER PERMISSION	Alters the state of a permission	"ALTER PERMISSION" on page 540
ALTER PROCEDURE (external)	Alters the description of an external procedure	"ALTER PROCEDURE (external)" on page 542
ALTER SEQUENCE	Alters the description of a sequence	"ALTER SEQUENCE" on page 543
ALTER TABLE	Alters the description of a table	"ALTER TABLE" on page 547
ALTER TRIGGER	Alters the description of a trigger	"ALTER TRIGGER" on page 578
COMMENT	Adds or replaces a comment to the description of an object	"COMMENT" on page 590
CREATE ALIAS	Creates an alias	"CREATE ALIAS" on page 605
CREATE FUNCTION	Creates a user-defined function (introduction)	"CREATE FUNCTION" on page 606
CREATE FUNCTION (external scalar)	Creates an external scalar function	"CREATE FUNCTION (external scalar)" on page 610
CREATE FUNCTION (external table)	Creates an external table function	"CREATE FUNCTION (external table)" on page 622
CREATE FUNCTION (sourced)	Creates a user-defined function based on another existing scalar or aggregate function	"CREATE FUNCTION (sourced)" on page 632
CREATE FUNCTION (SQL scalar)	Creates an SQL scalar function	"CREATE FUNCTION (SQL scalar)" on page 640
CREATE FUNCTION (SQL table)	Creates an SQL table function	"CREATE FUNCTION (SQL table)" on page 647
CREATE INDEX	Creates an index on a table	"CREATE INDEX" on page 653
CREATE MASK	Creates a column mask for a table	"CREATE MASK" on page 656
CREATE PERMISSION	Creates a row permission for a table	"CREATE PERMISSION" on page 662

Table 43. SQL Schema Statements (continued)

SQL Statement	Description	Reference
CREATE PROCEDURE	Creates a procedure (introduction)	"CREATE PROCEDURE" on page 666
CREATE PROCEDURE (external)	Creates an external procedure	"CREATE PROCEDURE (external)" on page 667
CREATE PROCEDURE (SQL)	Creates an SQL procedure	"CREATE PROCEDURE (SQL)" on page 675
CREATE SEQUENCE	Creates a sequence	"CREATE SEQUENCE" on page 682
CREATE TABLE	Creates a table	"CREATE TABLE" on page 688
CREATE TRIGGER	Creates a trigger	"CREATE TRIGGER" on page 718
CREATE TYPE	Creates a distinct type	"CREATE TYPE (distinct)" on page 734
CREATE VIEW	Creates a view of one or more tables or views	"CREATE VIEW" on page 743
DROP	Drops an object	"DROP" on page 776
GRANT (function or procedure privileges)	Grants privileges on a function or procedure	"GRANT (function or procedure privileges)" on page 793
GRANT (package privileges)	Grants privileges on a package	"GRANT (package privileges)" on page 797
GRANT (sequence privileges)	Grants privileges on a sequence	"GRANT (sequence privileges)" on page 799
GRANT (table or view privileges)	Grants privileges on a table or view	"GRANT (table or view privileges)" on page 801
GRANT (type privileges)	Grants privileges on a type	"GRANT (type privileges)" on page 804
RENAME	Renames a table	"RENAME" on page 849
REVOKE (function or procedure privileges)	Revokes privileges on a function or procedure	"REVOKE (function or procedure privileges)" on page 851
REVOKE (package privileges)	Revokes the privilege to execute statements in a package	"REVOKE (package privileges)" on page 855
REVOKE (sequence privileges)	Revokes privileges on a sequence	"REVOKE (sequence privileges)" on page 857
REVOKE (table or view privileges)	Revokes privileges on a table or view	"REVOKE (table and view privileges)" on page 859
REVOKE (type privileges)	Revokes the privilege to use a type	"REVOKE (type privileges)" on page 862

Table 44. SQL Data Change Statements

SQL Statement	Description	Reference
DELETE	Deletes one or more rows from a table	"DELETE" on page 764
INSERT	Inserts one or more rows into a table	"INSERT" on page 810
MERGE	Updates a target table using data from a source table	"MERGE" on page 820

Table 44. SQL Data Change Statements (continued)

SQL Statement	Description	Reference
TRUNCATE	Deletes all the rows from a table	"TRUNCATE" on page 892
UPDATE	Updates the values of one or more columns in one or more rows of a table	"UPDATE" on page 894

Table 45. SQL Data Statements

SQL Statement	Description	Reference
	All SQL Data Change statements	Table 44 on page 520
ALLOCATE CURSOR	Allocates a cursor for the result set identified by the result set locator variable	"ALLOCATE CURSOR" on page 529
ASSOCIATE LOCATORS	Gets the result set locator value for each result set returned by a procedure	"ASSOCIATE LOCATORS" on page 580
CLOSE	Closes a cursor	"CLOSE" on page 588
DECLARE CURSOR	Defines an SQL cursor	"DECLARE CURSOR" on page 749
FETCH	Positions a cursor on a row of the result table and assigns values from the row to variables	"FETCH" on page 789
FREE LOCATOR	Removes the association between a LOB locator variable and its value	"FREE LOCATOR" on page 792
LOCK TABLE	Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table	"LOCK TABLE" on page 819
OPEN	Opens a cursor	"OPEN" on page 828
REFRESH TABLE	Refreshes the data in a materialized query table	"REFRESH TABLE" on page 845
SELECT	Executes a query	"SELECT" on page 871
SELECT INTO	Assigns values to variables	"SELECT INTO" on page 872
SET transition-variable	Assigns values to transition variables in a trigger	"SET transition-variable" on page 888
SET variable	Assigns values to variables	"SET variable" on page 890
VALUES	Provides a way to invoke a user-defined function from a trigger	"VALUES" on page 902
VALUES INTO	Specifies a result table of no more than one row and assigns the values to variables	"VALUES INTO" on page 903

Table 46. SQL Transaction Statements

SQL Statement	Description	Reference
COMMIT	Ends a unit of work and commits the database changes made by that unit of work	"COMMIT" on page 596
RELEASE SAVEPOINT	Releases a savepoint within a unit of work	"RELEASE SAVEPOINT" on page 848
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 866
SAVEPOINT	Sets a savepoint within a unit of work	"SAVEPOINT" on page 869

Statements

Table 47. SQL Connection Statements

SQL Statement	Description	Reference
CONNECT (type 1)	Connects to a server and establishes the rules for remote unit of work	"CONNECT (type 1)" on page 598
CONNECT (type 2)	Connects to a server and establishes the rules for application-directed distributed unit of work	"CONNECT (type 2)" on page 602
RELEASE (connection)	Places one or more connections in the release-pending state	"RELEASE (connection)" on page 846
SET CONNECTION	Establishes the server of the process by identifying one of its existing connections	"SET CONNECTION" on page 875

Table 48. SQL Dynamic Statements

SQL Statement	Description	Reference
DESCRIBE	Describes the result columns of a prepared statement	"DESCRIBE" on page 769
DESCRIBE INPUT	Describes the input parameter markers of a prepared statement	"DESCRIBE INPUT" on page 773
EXECUTE	Executes a prepared SQL statement	"EXECUTE" on page 784
EXECUTE IMMEDIATE	Prepares and executes an SQL statement	"EXECUTE IMMEDIATE" on page 787
PREPARE	Prepares an SQL statement for execution	"PREPARE" on page 833

Table 49. SQL Session Statements

SQL Statement	Description	Reference
DECLARE GLOBAL TEMPORARY TABLE	Defines a declared temporary table	"DECLARE GLOBAL TEMPORARY TABLE" on page 755
SET CURRENT DECFLOAT ROUNDING MODE	Assigns a value to the CURRENT DECFLOAT ROUNDING MODE special register	"SET CURRENT DECFLOAT ROUNDING MODE" on page 877
SET CURRENT DEGREE	Assigns a value to the CURRENT DEGREE special register	"SET CURRENT DEGREE" on page 879
SET ENCRYPTION PASSWORD	Assigns a value to the default encryption password	"SET ENCRYPTION PASSWORD" on page 881
SET PATH	Assigns a value to the CURRENT PATH special register	"SET PATH" on page 883
SET SCHEMA	Assigns a value to the CURRENT SCHEMA special register	"SET SCHEMA" on page 886

Table 50. SQL Embedded Host Language Statements

SQL Statement	Description	Reference
BEGIN DECLARE SECTION	Marks the beginning of an SQL declare section	"BEGIN DECLARE SECTION" on page 582
CALL	Calls a procedure	"CALL" on page 584
END DECLARE SECTION	Marks the end of an SQL declare section	"END DECLARE SECTION" on page 783
INCLUDE	Inserts declarations into a source program	"INCLUDE" on page 808
WHENEVER	Defines actions to be taken on the basis of SQL return codes	"WHENEVER" on page 905

Table 51. SQL Control Statements

SQL Statement	Description	Reference
assignment-statement	Assigns a value to an output parameter or to a local variable	"assignment-statement" on page 918
CALL	Calls a procedure	"CALL statement" on page 920
CASE	Selects an execution path based on multiple conditions	"CASE statement" on page 922
compound-statement	Groups other statements together in an SQL routine	"compound-statement" on page 924
FOR	Executes a statement for each row of a table	"FOR statement" on page 933
GET DIAGNOSTICS	Obtains information about the previously executed SQL statement	"GET DIAGNOSTICS statement" on page 935
GOTO	Branches to a user-defined label within an SQL routine or trigger	"GOTO statement" on page 937
IF	Provides conditional execution based on the truth value of a condition	"IF statement" on page 939
ITERATE	Causes the flow of control to begin at the beginning of a labelled loop	"ITERATE statement" on page 941
LEAVE	Continues execution by leaving a block or loop	"LEAVE statement" on page 943
LOOP	Repeats the execution of a statement or group of statements	"LOOP statement" on page 944
REPEAT	Executes a statement or group of statements until a search condition is true	"REPEAT statement" on page 946
RESIGNAL	Resignals an error or warning condition	"RESIGNAL statement" on page 948
RETURN	Returns from a routine	"RETURN statement" on page 951
SIGNAL	Signals an error or warning condition	"SIGNAL statement" on page 953
WHILE	Repeats the execution of a statement while a specified condition is true	"WHILE statement" on page 956

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 or not the statement is executable.

An *executable statement* can be invoked in three ways:

- Embedded in an application program
- Dynamically prepared and executed
- Issued interactively.

Note: Statements embedded in REXX 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 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. 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 and COBOL, each embedded statement must be preceded by the keywords EXEC and SQL. For more information, see Chapter 15, "Coding SQL statements in C applications," on page 1063 and Chapter 16, "Coding SQL statements in COBOL applications," on page 1081.
- In Java, each embedded statement must be preceded by the keywords #sql. For more information, see Chapter 17, "Coding SQL statements in Java applications," on page 1099.
- In REXX, each embedded statement must be preceded by the keyword EXECSQL. For more information, see Chapter 18, "Coding SQL statements in REXX applications," on page 1117.

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. Thus, 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 bound (statically prepared). [106]

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*

106. In DB2 for z/OS, and DB2 for LUW, a program preparation option is available to allow reference to objects that do not exist when the SQL statements are bound.

executed, and acts as a no-operation if placed among executable statements of the application program. Therefore, do not follow such statements with 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).

In C, COBOL, and REXX, the statement can be prepared for execution by means of the (embedded) statement PREPARE and executed by means of 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 JDBC's 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 833 for rules concerning the parameter markers. When the prepared statement is executed, the parameter markers are effectively replaced by current values of the variables specified in the EXECUTE statement. See “EXECUTE” on page 784 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, and REXX, the successful or unsuccessful execution of the statement is indicated by the setting of an SQL return code after the EXECUTE (or EXECUTE IMMEDIATE) statement. Check the SQL return code as described above. See “SQL diagnostic information” on page 526 for more information. In Java, the unsuccessful execution of the statement is handled by Java Exceptions. For more information see “Handling SQL errors and warnings in Java” on page 1111.

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.

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 so constructed can be prepared for execution by means of the (embedded) statement PREPARE, and referenced by a (nonexecutable) statement DECLARE CURSOR. 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.

Used in this way, the *select-statement* must not contain references to variables. It can contain parameter markers instead. See "PREPARE" on page 833 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 828 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.

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 conditions and other information about the previously executed SQL statement are provided through the following mechanisms:

- For SQL routines, see "SQL-procedure-statement" on page 916.
- 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 SQL return codes:

- In C or COBOL, 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.

Use a stand-alone SQLSTATE to conform with the SQL 2011 Core standard.[107]

- In Java, for error conditions, the getSQLState method of JDBC's SQLException class can be used to get the SQLSTATE and the getErrorCode method can be used to get the SQLCODE. For more information, see "Handling SQL errors and warnings in Java" on page 1111.
- In REXX, an SQLCA is provided automatically.

SQLSTATE

The SQLSTATE is set by the database manager after execution of each SQL statement. 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 format of the SQLSTATE values is the same for all database managers and is consistent with the SQL 2011 Core standard. See Chapter 13, "SQLSTATE values—common return codes," on page 997 for more information and a complete list of the possible values of SQLSTATE.

SQLCODE

The SQLCODE is also set by the database manager after each SQL statement is executed 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.

SQLCODE values may provide additional product-specific information about an error or warning. Portable applications should use SQLSTATE values instead of SQLCODE values.

107. In DB2 for z/OS and DB2 for LUW, a program preparation option must be used to indicate the use of a stand-alone SQLCODE. In DB2 for z/OS, the same option must be used to indicate the use of a stand-alone SQLSTATE. For DB2 for z/OS use the precompiler option STDSQL(YES). For DB2 for LUW, use the program preparation option LANGLEVEL SQL92E.

SQL comments

In C and COBOL, static SQL statements can include host language or SQL comments. In Java and REXX, static SQL statements cannot include host language or SQL comments. For more information, see Chapter 17, “Coding SQL statements in Java applications,” on page 1099 and Chapter 18, “Coding SQL statements in REXX applications,” on page 1117.

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.

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 only be embedded in an SQL procedure. It is not an executable statement and cannot be dynamically prepared.

Authorization

None required.

Syntax

Description

cursor-name
Names the cursor. The name must not identify a cursor that has already been declared in the procedure.

CURSOR FOR RESULT SET *rs-locator-variable*
Specifies a result set locator variable that has been declared in the procedure 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 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 procedure.

Rules

- The following rules apply when you use an allocated cursor:
  - You cannot open an allocated cursor with the OPEN statement.
  - An allocated cursor cannot be used in a positioned UPDATE or DELETE 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.
- A commit operation closes allocated cursors that are not defined WITH HOLD by the procedure.
- 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;
```

ALTER FUNCTION (external)

The ALTER FUNCTION (external) statement alters an external 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

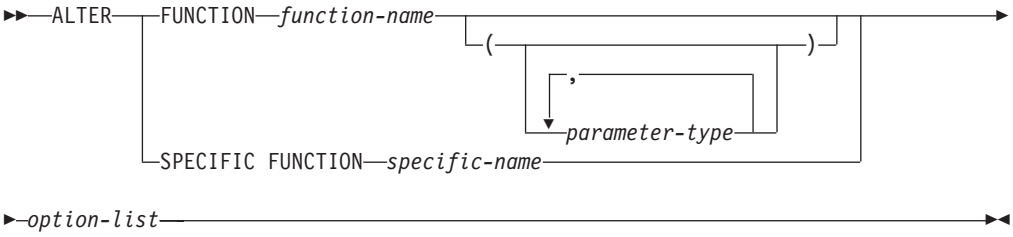
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the object
- Database administrator authority

If the SECURED option is specified or if the function is currently secure:

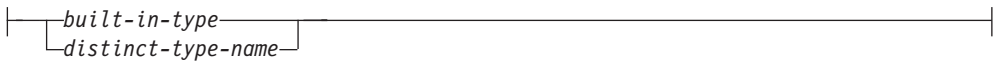
- The authorization ID of the statement must have Security administrator authority.

Syntax

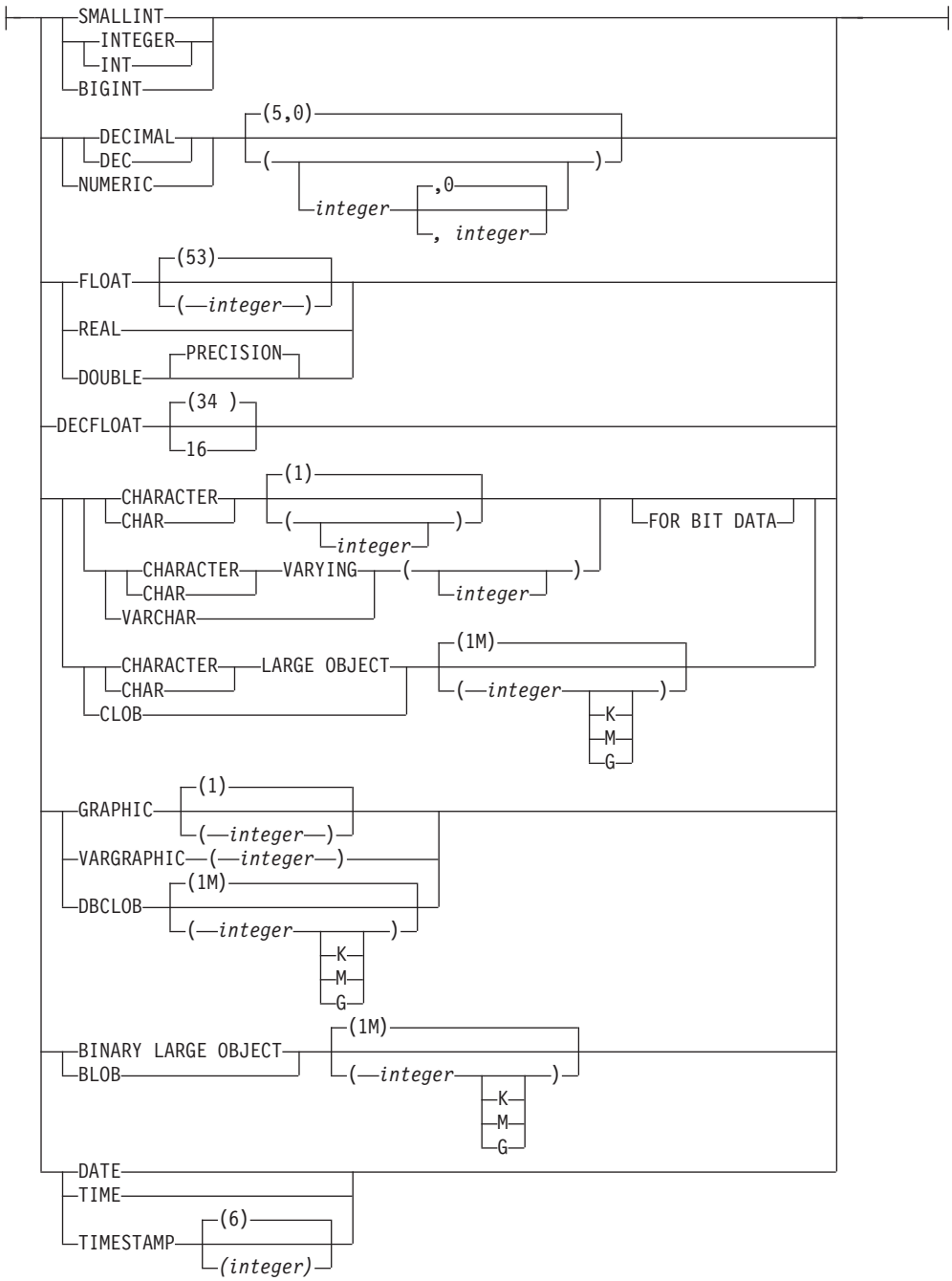


parameter-type:

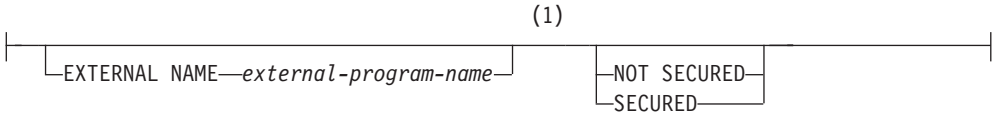
data-type:



built-in-type:



option-list:



Notes:

1 The clauses in the *option-list* can be specified in any order. Each clause can be specified at most once.

Description

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. The function can be identified by 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, DECIMAL() will be considered a match for a parameter of a function defined with a data type of DECIMAL(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 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.

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.

EXTERNAL NAME *external-program-name*
Specifies the program that will be executed when the function is invoked. The executable form of the external program need not exist when the ALTER FUNCTION statement is executed. However, it must exist at the current server when the function is invoked.

If a JAR is referenced then the JAR must exist when the external program name is specified.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row access control and column access control.

NOT SECURED
Specifies that the function is considered not secure for row access control and column access control. This is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

A function must be defined as secure when it is referenced in a row permission or a column mask.

Notes

General considerations for changing a function: See CREATE FUNCTION (External Scalar) and CREATE FUNCTION (External Table) for general information about defining a function. ALTER FUNCTION (external) allows individual attributes to be altered while preserving the privileges on the function.

Altering a function from NOT SECURED to SECURED: The function is considered secure after the ALTER FUNCTION statement is executed. The database manager treats the SECURED attribute as an assertion that declares that the user has established an audit procedure for all changes to the user-defined function. The database manager assumes that all subsequent ALTER FUNCTION statements are being reviewed through this audit process.

Invoking other user-defined functions in a secure function: When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, a user with Security administrator authority needs to ensure that those functions are allowed to access that data and should ensure that a change control audit procedure has been established for all changes to those functions.

Examples

Example 1: 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 (SQL)

The ALTER FUNCTION (SQL) statement alters an SQL function at the current server.

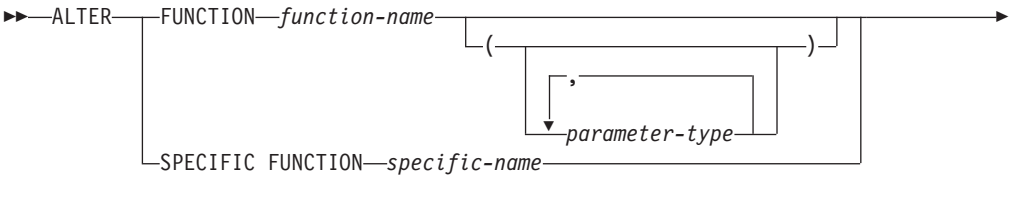
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

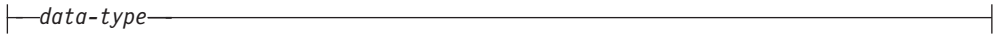
Authorization

The authorization ID of the statement must have security administrator authority.

Syntax

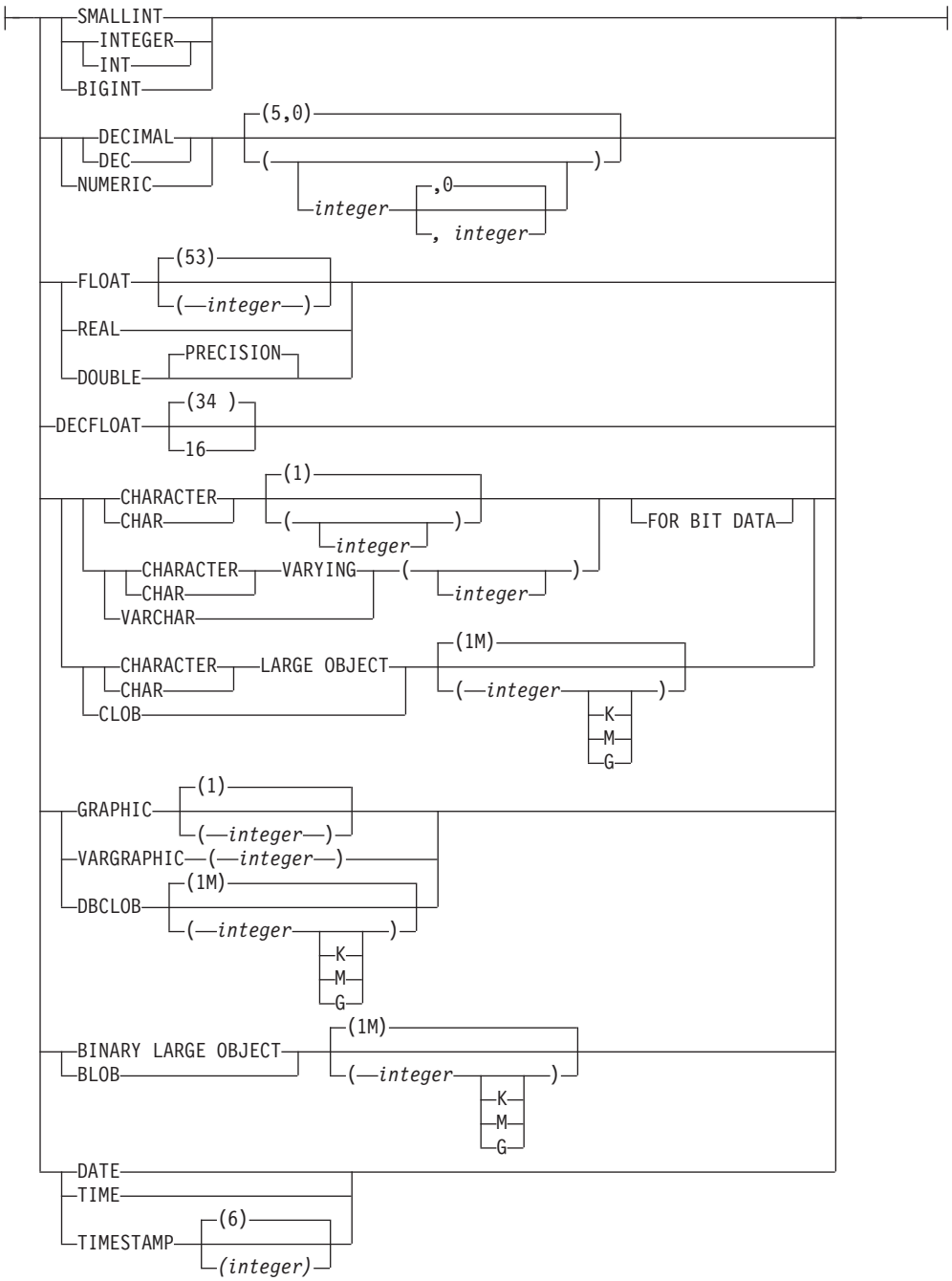


parameter-type:



data-type:

built-in-type:



option-list:

Description

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. The function can be identified by 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 user-defined type name is specified, the database manager searches the SQL path to resolve the schema name for the user-defined 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, DECIMAL() will be considered a match for a parameter of a function defined with a data type of DECIMAL(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 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.

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.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row access control and column access control.

NOT SECURED
Specifies that the function is considered not secure for row access control and column access control. This is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

A function must be defined as secure when it is referenced in a row permission or a column mask.

Notes

General considerations for changing a function: See CREATE FUNCTION (SQL scalar) and CREATE FUNCTION (SQL table) for general information about defining a function. ALTER FUNCTION (SQL) allows individual attributes to be altered while preserving the privileges on the function.

Altering a function from NOT SECURED to SECURED: The function is considered secure after the ALTER FUNCTION statement is executed. The database manager treats the SECURED attribute as an assertion that declares that the user has established an audit procedure for all changes to the user-defined function. The database manager assumes that all subsequent ALTER FUNCTION statements are being reviewed through this audit process.

Invoking other user-defined functions in a secure function: When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, a user with Security administrator authority needs to ensure that those functions are allowed to access that data and should ensure that a change control audit procedure has been established for all changes to those functions.

Examples

Example 1: Modify the definition for function MYFUNC to make the function secure for row and column access control.

```
ALTER FUNCTION MYFUNC
  SECURED
```

ALTER MASK

The ALTER MASK statement alters a column mask that exists 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 authorization ID of the statement must have security administrator authority.

Syntax

Description

mask-name
Identifies the column mask to be altered. It must identify a column mask that exists at the current server.

ENABLE or DISABLE

ENABLE
Specifies that the column mask is to be enabled for column access control. If column access control is not currently activated for the table, the column mask will become effective when column access control is activated for the table. If column access control is currently activated for the table, the column mask becomes effective immediately.

A column mask that gets an error when trying to be enabled cannot be enabled until any errors in the mask definition are resolved. This may require the column mask to be dropped and recreated with a modified definition.

ENABLE is ignored if the column mask is already defined as enabled for column access control.

DISABLE
Specifies that the column mask is to be disabled for column access control. If column access control is not currently activated for the table, the column mask will remain ineffective when column access control is activated for the table. If column access control is currently activated for the table, the column mask becomes ineffective.

DISABLE is ignored if the column mask is already defined as disabled for column access control.

Examples

Example 1: Enable column mask M1.

```
ALTER MASK M1 ENABLE
```

Example 2: Disable column mask M1.

```
ALTER MASK M1 DISABLE
```

ALTER PERMISSION

The ALTER PERMISSION statement alters a row permission that exists 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 authorization ID of the statement must have security administrator authority.

Syntax

Description

permission-name
Identifies the row permission to be altered. It must identify a row permission that exists at the current server. It cannot be a default permission.

ENABLE or DISABLE

ENABLE
Specifies that the row permission is to be enabled for row access control. If row access control is not currently activated for the table, the row permission will become effective when row access control is activated for the table. If row access control is currently activated for the table, the row permission becomes effective immediately.

A row permission that gets an error when trying to be enabled cannot be enabled until any errors in the permission definition are resolved. This may require the row permission to be dropped and recreated with a modified definition.

ENABLE is ignored if the row permission is already defined as enabled for row access control.

DISABLE
Specifies that the row permission is to be disabled for row access control. If row access control is not currently activated for the table, the row permission will remain ineffective when row access control is activated for the table. If row access control is currently activated for the table, the row permission becomes ineffective.

DISABLE is ignored if the row permission is already defined as disabled for row access control.

Examples

Example 1: Enable permission P1.

```
ALTER PERMISSION P1 ENABLE
```

Example 2: Disable permission P1.

```
ALTER PERMISSION P1 DISABLE
```

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:

- Ownership of the object
- Database administrator authority

Syntax

Description

procedure-name
Identifies the procedure by its name. The *procedure-name* must identify a procedure that exists at the current server.

EXTERNAL NAME *external-program-name*
Specifies the program that will be executed when the procedure is called by the CALL statement. The executable form of the external program need not exist when the ALTER PROCEDURE statement is executed. However, it must exist at the current server when the procedure is called.

If a JAR is referenced then the JAR must exist when the external program name is specified.

Notes

General considerations for changing a procedure: See CREATE PROCEDURE for general information on defining a procedure. ALTER PROCEDURE (external) allows individual attributes to be altered while preserving the privileges on the procedure.

Examples

Example 1: Modify the definition for procedure MYPROC to change the name of the external program that will be invoked when the procedure is called. The name of the external program is PROG10A.

```
ALTER PROCEDURE MYPROC
  EXTERNAL NAME PROG10A
```

ALTER 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

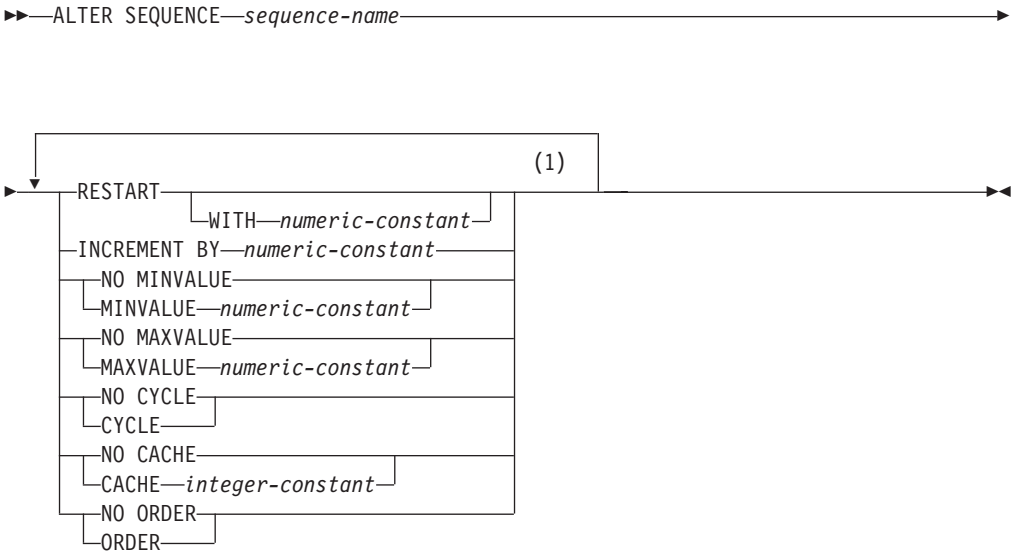
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- ALTER privilege on the sequence to be altered
- Database administrator authority

Syntax



Notes:

1 The same clause must not be specified more than once.

Description

sequence-name
Identifies the sequence to be altered. The *sequence-name* must identify a sequence that exists at the current server. *sequence-name* must not be a sequence generated by the system for an identity column.

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 existing 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. The value must not exceed the value of a large integer constant and must not contain nonzero digits 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 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 nonzero digits existing 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 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. 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 nonzero digits existing 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 the sequence should continue to generate values after reaching either its maximum or minimum value.

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 its maximum value, it generates its minimum value; or after a descending sequence reaches its minimum value, 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, 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.

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 a 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 the case of a system failure, shutdown or database deactivation. 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 a synchronous write.

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.
- The data type of a sequence cannot be changed. Instead, drop and recreate the sequence specifying the desired data type for the new sequence.
- All 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.

Examples

Example 1: 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 the 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) DEFINITION ONLY
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:

- The ALTER privilege for the table
- Database administrator authority

If defining 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 on the table
- The REFERENCES privilege on each column of the specified parent key
- Database administrator authority

If dropping the primary key of table T, the privileges held by the authorization ID of the statement must include at least one of the following on every table that is a dependent of T:

- The ALTER privilege on the table
- Database administrator authority

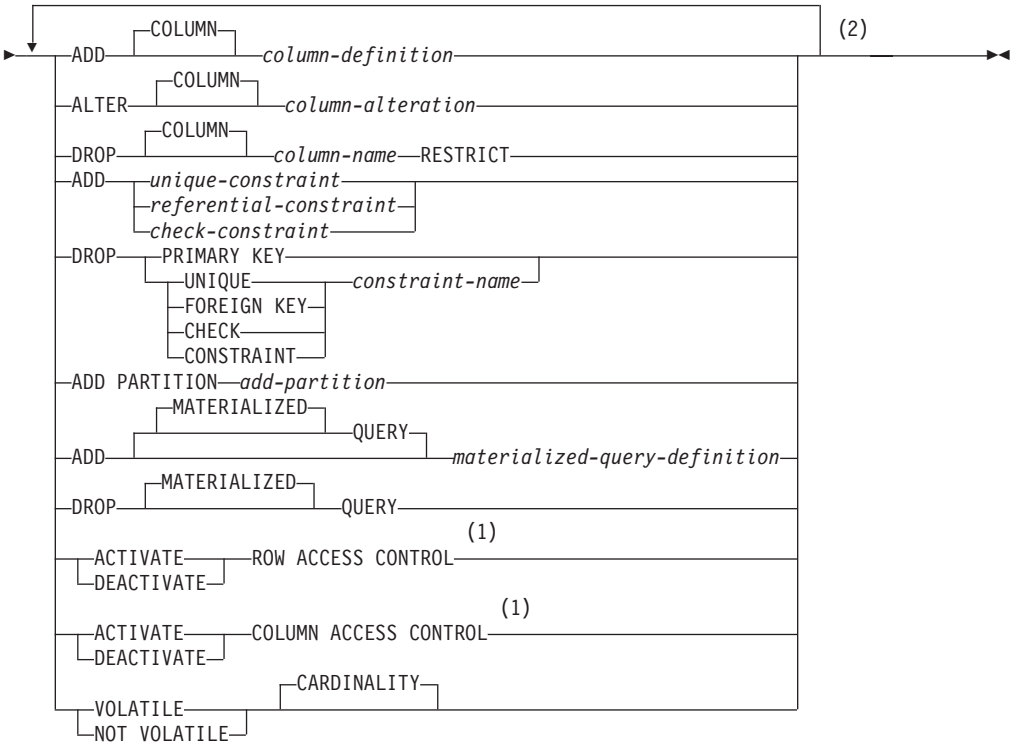
If an ACTIVATE or DEACTIVATE clause is specified:

- The authorization ID of the statement must have Security administrator authority.

If referring to a distinct type, the privileges held by the authorization ID of the statement must include at least one of the following:

- USAGE privilege on the distinct type
- Database administrator authority

Syntax

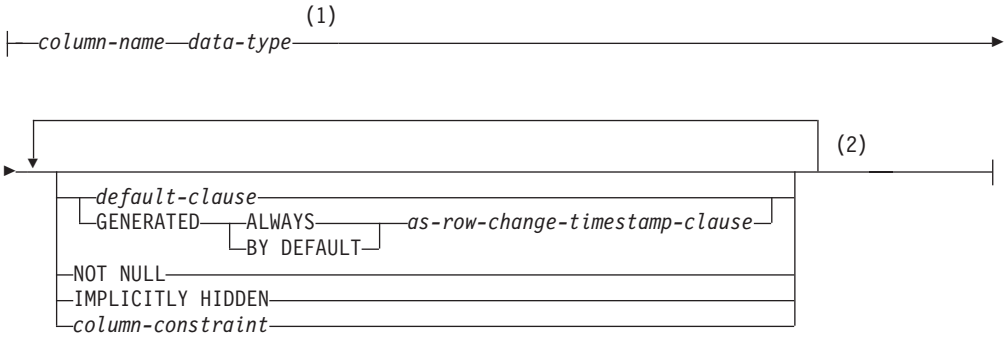


Notes:

1 If an ACTIVATE or DEACTIVATE clause is specified, clauses other than ACTIVATE or DEACTIVATE are not allowed on the ALTER TABLE statement. Each clause must not be specified more than once.

2 The same clause must not be specified more than once, except for the ALTER COLUMN clause, which can be specified more than once. Do not specify DROP CONSTRAINT if DROP FOREIGN KEY or DROP CHECK is specified.

column-definition:



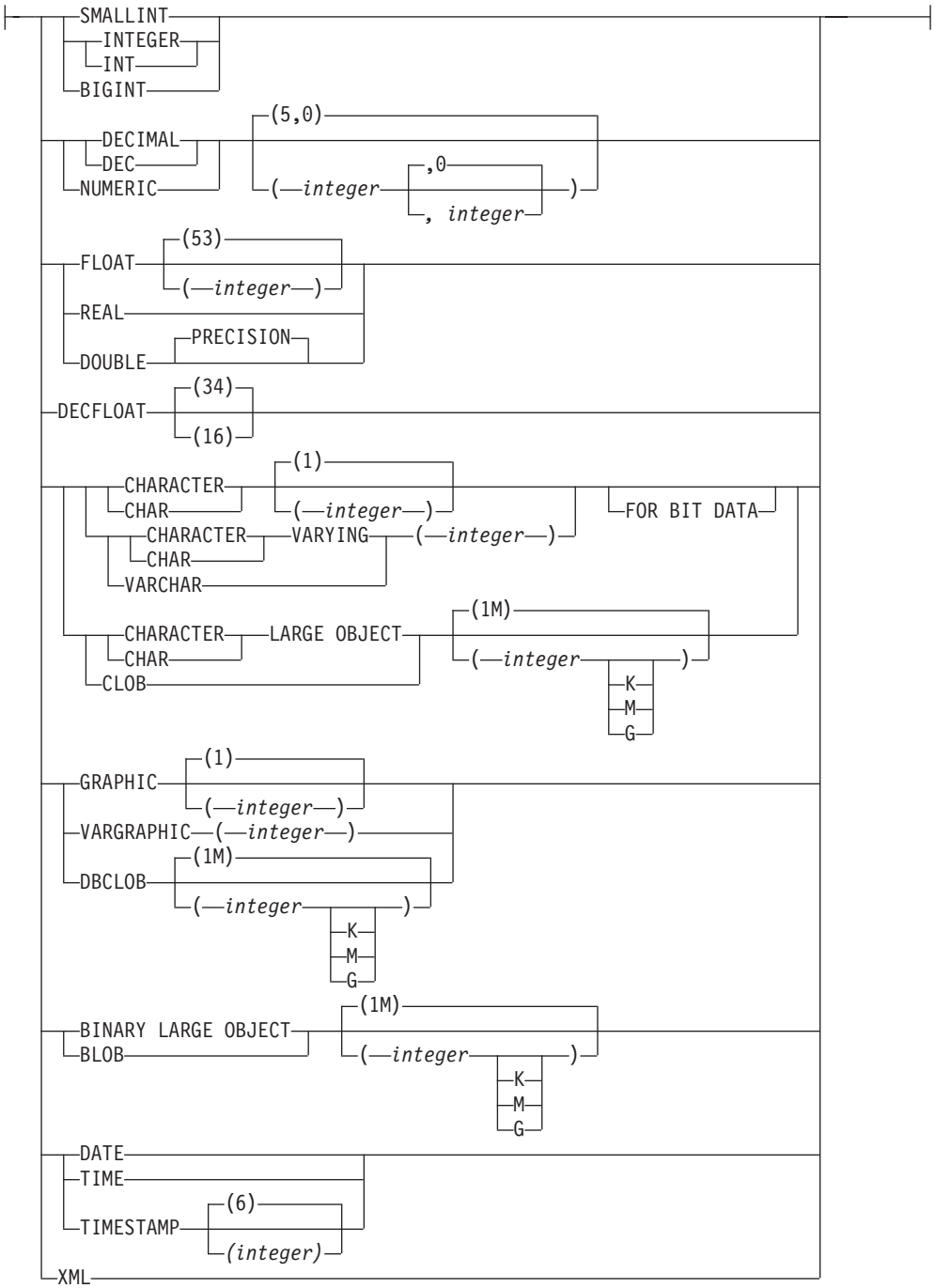
Notes:

1 *data-type* is optional if *as-row-change-timestamp-clause* is specified

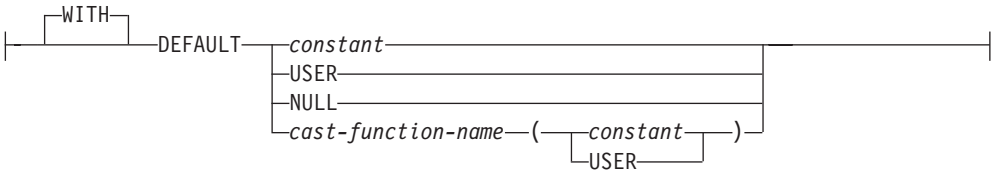
2 The same clause must not be specified more than once.

data-type:

built-in-type:

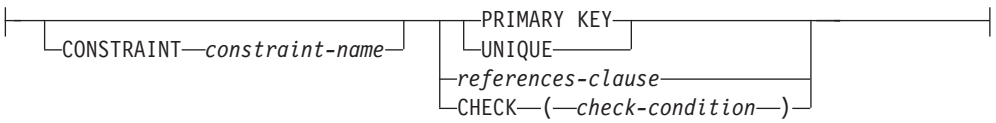


default-clause:

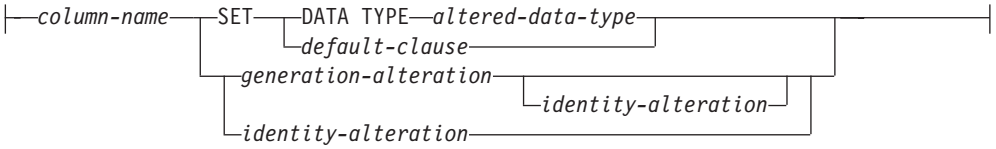


as-row-change-timestamp-clause:

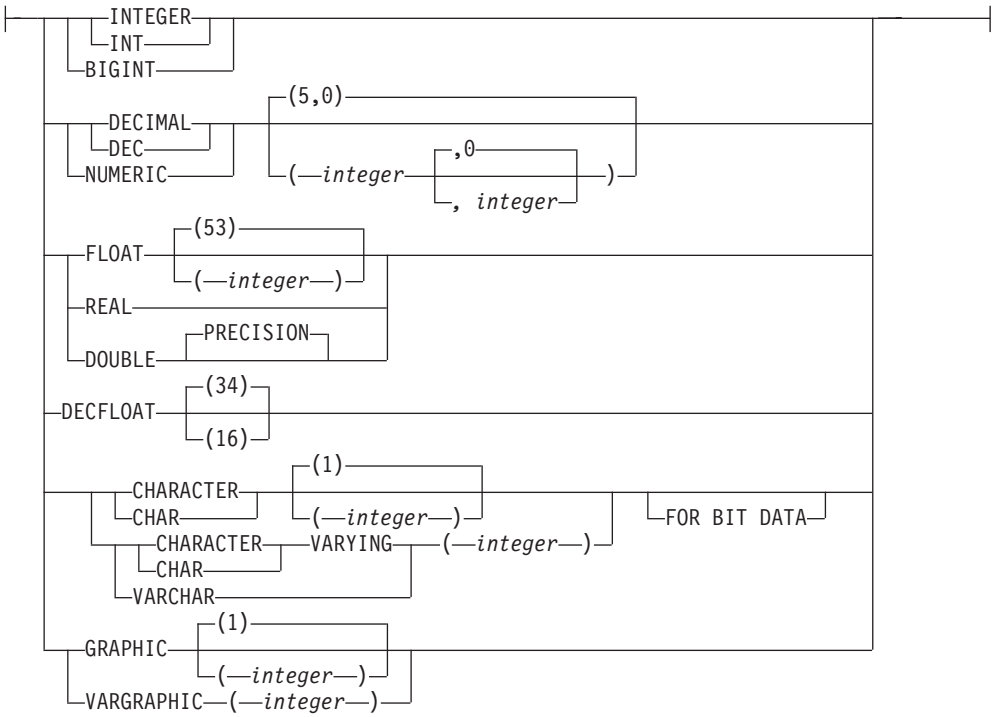
column-constraint:



column-alteration:

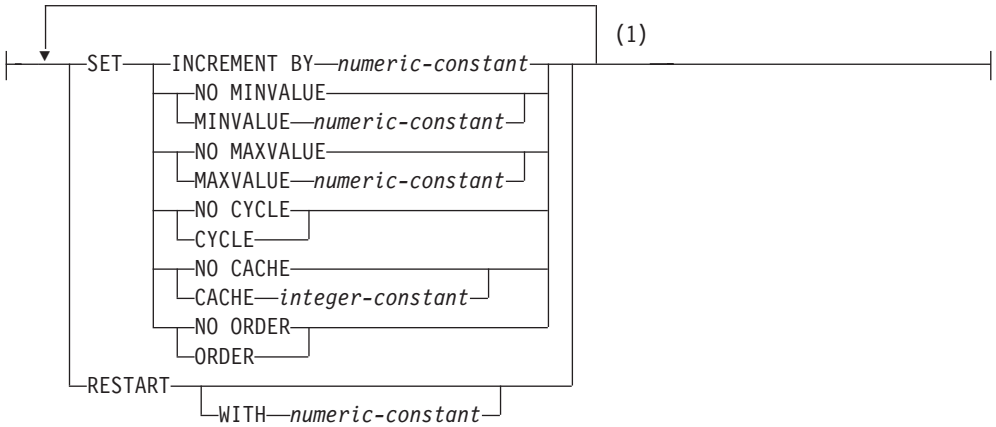


altered-data-type:



generation-alteration:

identity-alteration:



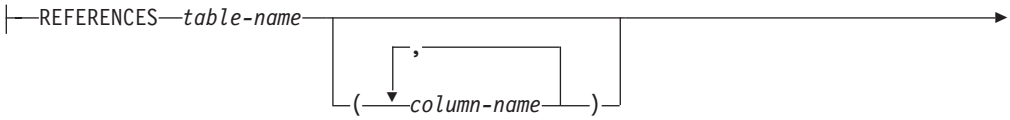
Notes:

1 The same clause must not be specified more than once.

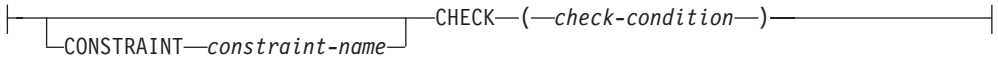
unique-constraint:

referential-constraint:

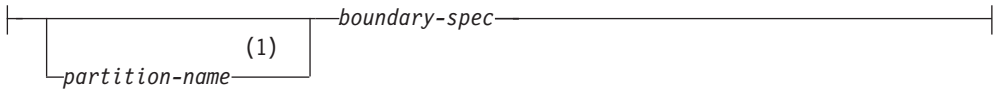
references-clause:



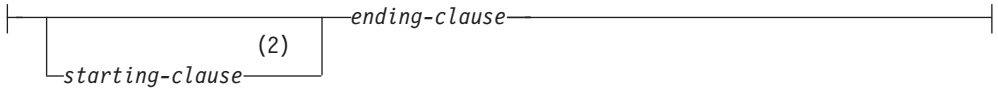
check-constraint:



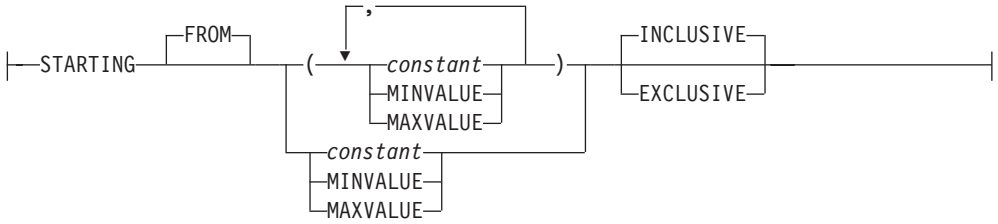
add-partition:



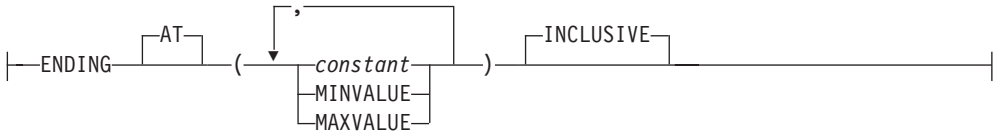
boundary-spec:



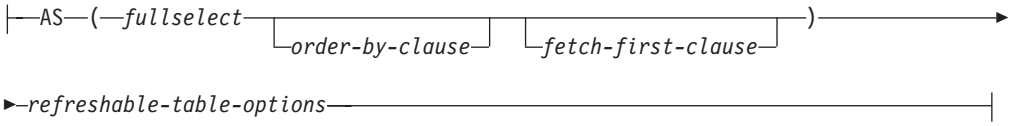
starting-clause:



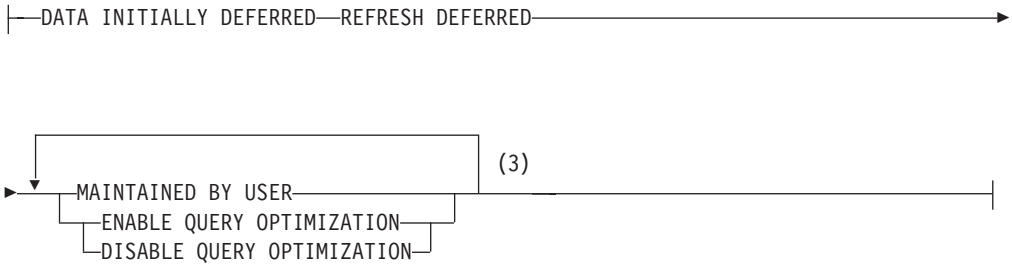
ending-clause:



materialized-query-definition:



refreshable-table-options:



Notes:

1 In DB2 for z/OS, *partition-name* must not be specified.

2 In DB2 for z/OS, the *starting-clause* must not be specified. In DB2 for i, the *starting-clause* is optional. In DB2 for LUW, the first *partition-element* must include a *starting-clause*.

3 The same clause must not be specified more than once. MAINTAINED BY USER must be specified.

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, alterations are limited to adding or dropping the materialized query table, and adding or dropping RESTRICT ON DROP.

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 row change timestamp. If the table previously had *n* columns, the ordinality of the new column is *n*+1. The value of *n*+1 must not exceed 750. [108] See Table 63 on page 967 for more information.

A table can have only one row change timestamp.

Adding the new column must not make the total byte count of all columns exceed the maximum record size. The maximum record size is 32 677. See Table 63 on page 967 for more information.

column-name
Names the column to be added to the table. Do not use the same name for more than one column name of the table. Do not qualify *column-name*.

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 688 for the description of built-in types.

108. This value is 1 less if the table is a dependent table.

distinct-type-name
Specifies 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 in the SQL path.

DEFAULT
Specifies a default value for the column. This clause must not be specified more than once in the same *column-definition*. DEFAULT cannot be specified for the following types of columns :

- An identity column (the database manager generates default values)
- A row change timestamp column (the database manager generates default values)
- An XML column

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 77. A 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 an SQL data change statement as the default for the column. The data type of the column or 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 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 was specified, DEFAULT NULL must not be specified within the same *column-definition*.

cast-function-name
Specifies the name of the cast function that matches the name of the distinct type name of the data type for the column.

The schema name of the cast function, whether it is explicitly specified or implicitly resolved through function resolution, must be the same as the explicitly or implicitly specified schema name of the distinct type. This form of the DEFAULT value can only be used with columns that are defined as a 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.

USER
Specifies the value of the USER special register at the time of an SQL data change statement as the default for the column. 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

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.

If the value specified is not valid, an error is returned.

GENERATED
Specifies that the database manager generates values for the column. GENERATED must be specified if the column is to be considered 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 a 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 row change timestamp column has a unique constraint or a unique index that solely specifies the row change timestamp column.

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. 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.

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*
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.

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. PRIMARY KEY is not supported by DB2 for z/OS for *column-constraint*.

The NOT NULL clause must be specified with this 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. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

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. UNIQUE is not supported by DB2 for z/OS for *column-constraint*.

The NOT NULL clause must be specified with this clause. This clause cannot be specified more than once in a column definition and must not be specified if the PRIMARY KEY clause is specified in the column definition. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

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. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML
- A row change timestamp

For more information, see REFERENCES clause.

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. For more information, see CHECK clause.

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.

A column can only be referenced once in an ALTER 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. A column cannot be altered if any of the following conditions are true:

- The table is used in a materialized query table definition
- The column is referenced in a referential constraint
- The column is a LOB column

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 in the same ALTER TABLE statement.

SET DATA TYPE *altered-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 on built-in data types, see "CREATE TABLE" on page 688.

G Table 52 lists the compatible data types. For DB2 for z/OS and DB2 for LUW, a
G data type alteration may require a table reorganization before a table can again
G be accessed. See the product references for more information.

G For DB2 for z/OS, when a table partition key column is altered, see the DB2
G for z/OS SQL Reference manual for considerations on the limit key value.

The data type of an identity column cannot be altered.

The data type of a string column used in a partitioning key cannot be altered.

The specified length, precision, or scale must not be less than the existing length, precision, or scale.

Table 52. Compatible Data Types

From type	To type
SMALLINT	INTEGER, BIGINT, DECIMAL (*q*, *t*); *q*-*t* > 4, REAL, DOUBLE
INTEGER	BIGINT, DECIMAL (*q*, *t*); *q*-*t* > 9, DOUBLE
BIGINT	DECIMAL (*q*, *t*); *q*-*t* > 18
REAL	DOUBLE
DECIMAL (*p*, *s*)	DECIMAL (*q*, *t*); *q* >= *p*; *t* >= *s*; (*q*-*t*) >= (*p*-*s*)
CHARACTER (*n*)	CHARACTER (*n*+*x*), VARCHAR (*n*+*x*)
VARCHAR (*n*)	CHARACTER (*n*+*x*), VARCHAR (*n*+*x*)
GRAPHIC (*n*)	GRAPHIC (*n*+*x*), VARGRAPHIC (*n*+*x*)
VARGRAPHIC (*n*)	VARGRAPHIC (*n*+*x*), GRAPHIC (*n*+*x*)

Altering the column must not make the total byte count of all columns exceed the maximum record size. The maximum record size is 32 677. If the column is in the table partitioning key, the new partitioning key cannot exceed 255-n.See Table 63 on page 967 for more information.

If the column is used in a unique constraint or an index, the new length must not cause the sum of the stored lengths for the unique constraint or index to exceed 2000. See Table 63 on page 967 for more information.

Changing the attributes will cause any existing values in the column to be converted to the new column attributes using storage assignment rules.

Altering the data type attributes of a column can affect a row permission or column mask that is defined for the table. When data type attributes of a column change, row permissions and column masks are reevaluated using the new column attributes. If an error is encountered during the reevaluation process, the ALTER statement fails.

If a row permission or a column mask defined on a different table references this column, the row permission or column mask will not be reevaluated until it is used or is the object of an ALTER REGENERATE. A reevaluation error will result in failure of either the regenerate or the statement that first requires use of the column mask or row permission. The row permission or column mask may need to be dropped and recreated to fix the error.

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. See "CREATE TABLE" on page 688 for a description of the attributes.

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 non-zero digits existing to the right of the decimal point.

DROP COLUMN

Drops the identified column from the table. A column cannot be dropped if any of the following conditions are true:

- The table contains check constraints
- The table is referenced in a materialized query table definition
- A view that is dependent on the table has INSTEAD OF triggers
- A trigger is defined on the table
- A row permission or column mask is dependent on 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 the only column of a table. The name must not identify a column that

was already added or altered in this ALTER TABLE statement. The specified column cannot be dropped if any of the following conditions are true:

- The column is an XML column
- The column is part of the table partitioning key
- All of the remaining columns in the table are hidden

RESTRICT
Specifies that the column cannot be dropped if any views, indexes, triggers, constraints, materialized query tables, or global variables are dependent on the column.

ADD unique-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.

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. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

The number of identified columns must not exceed 64 and the sum of their length attributes must not exceed 2000. See Table 63 on page 967 for more information.

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. The identified columns must be defined as NOT NULL. Any existing values in the set of columns must be unique.

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 constraint.

G In DB2 for z/OS, if the table is in a table space that is implicitly created, and
G no unique index is defined on the identified columns, a unique index will
G automatically be created to enforce the unique key constraint. Otherwise, the
G unique index must already exist.

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. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

The number of identified columns must not exceed 64 and the sum of their length attributes must not exceed 2000. See Table 63 on page 967 for more information. The table must not already have a primary key.

The identified columns cannot be the same as the set of 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. The identified columns must be defined as NOT NULL. Any existing values in the set of columns must be unique.

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.

G In DB2 for z/OS, if the table is in a table space that is implicitly created, and
G no unique index is defined on the identified columns, a primary index will
G automatically be created. Otherwise, the unique index must already exist.

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.

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. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML
- A row change timestamp

The number of identified columns must not exceed 64 and the sum of their length attributes must not exceed 2000. See Table 63 on page 967 for more information.

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, or a declared temporary table.

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
G constraints are allowed, but not recommended. In DB2 for z/OS, duplicate
G referential constraints are ignored with a warning.

Let T2 denote the identified parent table.

G In DB2 for z/OS, if T1 and T2 are the same table, ON DELETE CASCADE
G or ON DELETE NO ACTION must be specified.

(*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 data type of the column must not be a LOB data type or a distinct type based on a LOB data type or a row change timestamp. The number of identified columns must not exceed 64 and the sum of their length attributes must not exceed 2000. See Table 63 on page 967 for more information.

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 table must have a unique index with a key that is, respectively, identical to the primary key or the UNIQUE constraint. The keys are identical only if they have the same number of columns and the *n*th column name of one is the same as the *n*th column name of the other. 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 and other attributes.

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. In DB2 for z/OS, the table space of a non-empty table is placed in a check pending status. Therefore, every non-null 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 four possible actions:

- NO ACTION (default) [109]
- RESTRICT
- CASCADE
- SET NULL

SET NULL must not be specified unless some column of the foreign key allows null values. SET NULL must not be specified if T1 has an update trigger.

CASCADE must not be specified if T1 has a delete trigger.

In DB2 for LUW and DB2 for z/OS, a self-referencing table with a SET NULL or RESTRICT rule must not be a dependent in a referential constraint with a delete rule of CASCADE.

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.

109. In DB2 for z/OS, the default depends on the value of the CURRENT RULES special register when the CREATE TABLE statement is processed. If the value of the register is 'DB2', the default is RESTRICT. If the value is 'SQL', the default is NO ACTION.

- If RESTRICT or NO ACTION is specified, an error is returned 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.

A cycle involving two or more tables must not cause a table to be delete-connected to itself unless all of the delete rules in the cycle are CASCADE. Thus, if the relationship would form a cycle and T2 is already delete-connected to T1, then the constraint can only be defined if it has a delete rule of CASCADE and all other delete rules of the cycle are CASCADE.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. Let T3 denote a table identified in another FOREIGN KEY clause (if any) of the CREATE TABLE statement. The delete rules of the relationships involving T2 and T3 must be the same and must not be SET NULL if:

- T2 and T3 are the same table, or
- T2 is a descendant of T3 and the deletion of rows from T3 cascades to T2, or
- T3 is a descendant of T2 and the deletion of rows from T2 cascades to T3, or
- T2 and T3 are both descendants of the same table and the deletion of rows from that table cascades to both T2 and T3,

If *r* is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as *r*.

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. [110]

The *check-condition* is a form of the *search-condition*, except:

- Columns with the following data types cannot be specified:
  - DECFLOAT or a distinct type based on a DECFLOAT
  - A LOB data type or a distinct type based on a LOB data type
  - XML
- It can be up to 3800 bytes long, not including redundant blanks. See Table 63 on page 967 for more information.

110. In DB2 for z/OS, the value of the CURRENT RULES special register must be 'STD' to get this behavior.

- It must not contain any of the following:
  - Subqueries
  - Built-in functions
  - Aggregate functions
  - Variables
  - Parameter markers
  - *sequence-reference*s
  - OLAP specifications
  - Special registers
  - User-defined functions (except cast functions generated for distinct types)
  - CASE expressions

G In DB2 for z/OS, the *check-condition* is subject to additional restrictions. See the
G product reference for further information.

For more information about *search-condition*, see "Search conditions" on page 178.

DROP

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.

If a primary index was implicitly created to support uniqueness of the primary key, it is dropped.

DROP 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.

If a unique index was implicitly created to support uniqueness of the unique constraint, it is dropped.

DROP 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.

DROP CHECK *constraint-name*

Drops the check constraint *constraint-name*. The *constraint-name* must identify a check constraint on the table.

DROP CONSTRAINT *constraint-name*

Drops the constraint *constraint-name*. The *constraint-name* must identify a primary key, unique, referential, or check constraint in the table. If the constraint is a PRIMARY KEY or UNIQUE constraint, all referential constraints in which the primary key or unique constraint is a parent are also dropped.

DROP CONSTRAINT must not be used in the same ALTER TABLE statement as DROP PRIMARY KEY, DROP UNIQUE, DROP FOREIGN KEY or DROP CHECK.

ADD PARTITION add-partition

Adds one or more data partitions to a partitioned table. The specified table must be a partitioned table. Adding a partition is not allowed if the table is a materialized query table or a materialized query table is defined on the table. The number of data partitions must not exceed 256.

partition-name
Names the data partition. The name must not be the same as any other data partition for the table. In DB2 for z/OS *partition-name* must not be specified.

boundary-spec
Specifies the boundaries of a range partition. The specified table must be a range partitioned table. See "CREATE TABLE" on page 688 for a description of the *boundary-spec*.

ADD MATERIALIZED QUERY materialized-query-definition

Changes a base table to a materialized query table. The table specified by *table-name* must not:

- be already defined as a materialized query table
- have any primary keys, unique constraints (unique indexes), referential constraints (foreign keys), check constraints, or triggers defined
- be referenced in the definition of another materialized query table
- be directly or indirectly referenced in the fullselect

fullselect

Defines the query in which the table is based. The columns of the existing table must:

- have the same number of columns
- have exactly the same column definitions
- have the same column names in the same ordinal positions

as the result columns of the *fullselect*. The *fullselect* 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 details about specifying the *fullselect* for a materialized query table, see "CREATE TABLE" on page 688.

order-by-clause

Specifies the ordering of the rows of the result table of the *fullselect*. See "order-by-clause" on page 494.

fetch-first-clause

Specifies the maximum number of rows that can be retrieved by the *fullselect*. See "fetch-first-clause" on page 497.

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.

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.

G ADD MATERIALIZED QUERY *materialized-query-definition* is not supported by DB2
G for LUW, but the same functionality can be obtained using SET MATERIALIZED
G QUERY AS *materialized-query-definition*.

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 COLUMN ACCESS CONTROL or DEACTIVATE COLUMN ACCESS CONTROL

Specifies whether enabled column masks are to be applied by the database manager to mask column values returned from the table.

ACTIVATE COLUMN ACCESS CONTROL

Specifies to activate column access control for the table. If the table is an alias, column access control is activated for the base table.

The table must not be referenced in the definition of a view if:

- the view is defined with the WITH CHECK OPTION clause where the search conditions from the view or from the underlying views are checked during the insert or update operation.
- an INSTEAD OF trigger that is defined with the NOT SECURED attribute exists for the view.

The access to the table is not restricted but when the table is referenced in a data manipulation statement, all enabled column masks that have been created for the table are applied by the database manager to mask the values returned for the columns referenced in the final result table of the queries or to determine the new values used in the data change statements. A column mask

that gets an error when trying to be activated cannot be activated until any errors in the mask definition are resolved. This may require the column mask to be dropped and recreated with a modified definition.

If a materialized query table that depends on the table (directly, or indirectly through a view) for which column level access control is being activated and that materialized query table does not already have row level access control activated, row level access control is implicitly activated for the materialized query table. This restricts direct access to the contents of the materialized query table. A query that explicitly references the table before such a row permission is defined will return a warning that there is no data in the table. To provide access to the materialized query table, an appropriate row permission can be created, or an ALTER TABLE DEACTIVATE ROW ACCESS CONTROL on the materialized query table can be issued to remove the row level protection if that is appropriate.

ACTIVATE COLUMN ACCESS CONTROL is ignored if column access control is already defined as activated for the table.

DEACTIVATE COLUMN ACCESS CONTROL
Specifies to deactivate column access control for the table. When the table is referenced in a data manipulation statement, any enabled column masks defined on the table are not applied by the database manager to control the values returned for the columns referenced in the final result table of the queries or to determine whether the new values can be used in the data change statements.

DEACTIVATE COLUMN ACCESS CONTROL is ignored if column access control is already defined as not activated for the table.

VOLATILE CARDINALITY or NOT VOLATILE CARDINALITY

Indicates 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 *table-name* can vary significantly at run time, from empty to large. An index will be used to access the table, 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 statistics of the table at the time the access plan is built.

Notes

Column references: A column can only be referenced once in an ADD COLUMN or ALTER 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.

Columns not automatically added to views: Any columns added via ALTER TABLE will not automatically be added to any existing view of the table.

Adding a row change timestamp column: When you add a row change timestamp column to an existing table, when the initial values for existing rows are stored is product-specific.

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.

Cascaded effects: Altering a table can have effects on dependent objects.

- Invalidation of access plans and packages: Adding or dropping primary, foreign or unique keys or check constraints or altering column lengths may invalidate access plans. The rules are product-specific. Adding a data partition will cause any packages dependent on the table to be invalidated. Altering a table to change it from a regular base table to a materialized query table with REFRESH DEFERRED will cause any packages dependent on the table to be invalidated. Altering a table to change it from a materialized query table to a regular base table will cause any packages dependent on the table to be invalidated.
- Other cascaded effects of the ALTER TABLE statement are product-specific.

Functions in the SYSFUN schema: For DB2 for LUW, VARCHAR and VARGRAPHIC columns that have been altered to be greater than 4000 and 2000, respectively, must not be used as input parameters in functions in the SYSFUN schema.

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.

Names of indexes or constraints created automatically: The method used to automatically generate the name of an index or a constraint that is created during execution of the ALTER TABLE statement is product specific.

Order of operations: The order of operations within an ALTER TABLE statement is product-specific.

Considerations for using row access control and column access control:

Row access control that is activated explicitly: The ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for a table. When this happens, a default row permission is implicitly created and allows no access to any rows of the table, unless another enabled row permission exists that provides access for the authorization IDs that are specified in the definition of the permission. The default row permission is always enabled.

When the table is referenced in a data manipulation statement, all enabled row permissions that have been created for the table, including the default row permission, are implicitly applied by DB2 to control which rows in the table are accessible. A row access control search condition is derived by application of the logical OR operator to the search condition in each enabled row permission. This derived search condition acts as a filter to the table before any user specified operations such as predicates, grouping, ordering, etc. are processed. This derived search condition permits the authorization IDs that are specified in the permission definitions to access certain rows in the table. See the description of subselect for information on how the application of enabled row permissions affects the fetch

operation. See the data change statements for information on how the application of enabled row permissions affects the data change operation.

Row access control remains enforced until the DEACTIVATE ROW ACCESS CONTROL clause is used to stop enforcing it.

Implicit row permission that is created when row access control is activated for a table: When the ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for a table, DB2 implicitly creates a default row permission for the table. The default row permission prevents all access to the table. The implicitly created row permission is in the same schema of the base table and has a product-specific name.

The default row permission is always enabled.

The default row permission is dropped when row access control is deactivated or when the table is dropped.

Activating column access control: The ACTIVATE COLUMN ACCESS CONTROL clause is used to activate column access control for a table. The access to the table is not restricted but when the table is referenced in a data manipulation statement, all enabled column masks that have been created for the table are applied to mask the column values referenced in the final result table of the query or to determine the new values used in the data change statements.

When column masks are used to mask the column values, they determine the values in the final result table. If a column has a column mask and the column (a simple reference to a column name or embedded in an expression) appears in the outermost select list, the column mask is applied to the column to produce the values for the final result table. If the column does not appear in the outermost select list but it participates in the final result table, for example, it appears in a materialized table expression or view, the column mask is applied to the column in such a way that the masked value is included in the result table of the materialized table expression or view so that it can be used in the final result table.

The application of column masks does not interfere with the operations of other clauses within the statement such as the WHERE, GROUP BY, HAVING, SELECT DISTINCT, and ORDER BY. The rows returned in the final result table remain the same, except that the values in the resultant rows might have been masked by the column masks. As such, if the masked column also appears in an ORDER BY sort-key, the order is based on the original column values and the masked values in the final result table might not reflect that order. Similarly, the masked values might not reflect the uniqueness enforced by SELECT DISTINCT. If the masked column is embedded in an expression, the result of the expression can become different because the column mask is applied on the column before the expression evaluation can take place. For example, applying a column mask on column SSN can change the result of aggregate function COUNT(DISTINCT SSN) because the DISTINCT operation is performed on the masked values. On the other hand, if the expression in a query is the same as the expression used to mask the column value in the column mask definition, the result of the expression in the query might remain unchanged. For example, the expression in the query is 'XXX-XX-' || SUBSTR(SSN, 8, 4) and the same expression appears in the column mask definition. In this particular example, the user can replace the expression in the query with column SSN to avoid the same expression gets evaluated twice.

The following are the contexts where the column masks are used by DB2 to mask the column values for the result of a query. Certain restrictions might apply to some contexts.

- the outermost SELECT clause of a SELECT or SELECT INTO statement, or if the column does not appear in the outermost select list but it participates in the final result table, the outermost SELECT clause of the corresponding materialized table expression or view where the column appears.
- the outermost SELECT clause that is used to derive the new values for an INSERT, UPDATE, or MERGE statement, or a SET transition-variable assignment statement
- the same applies to a scalar-fullselect expression that does not use set operators and appears in the outermost SELECT clause of the above statements, the right side of a SET variable assignment statement, the VALUES INTO statement, or the VALUES statement.
- the same applies to the SQL statements or the equivalences such as the assignment statement that appear in an SQL procedure or SQL function.

A column mask is created without knowing all of the contexts in which it might be used. To mask a column value in the final result table, the column mask definition is merged into the statement by DB2. When the column mask definition is brought into the context of the statement, it might conflict with certain SQL semantics in the statement. Therefore, in some situations the combination of the statement and the application of a column mask can return an error. The following describes when the error might be returned:

1. The column masks cannot be applied to the columns in the select lists that derive the final result table of set operations because one of the set operators that is used to derive the final result table is UNION ALL, UNION, EXCEPT, or INTERSECT.

   The rows in the final result table are derived from multiple result tables (R1 and R2) combined by the set operators. DB2 does not guarantee which rows are derived from which result table (R1 or R2). Therefore, the rows in the final result table can vary each time if the column masks are applied but one result table (R1) has column masks and the other result table (R2) does not have column masks or has different column masks. DB2 returns an error without checking whether the columns in the final result table rows have the same column masks.
2. The column mask cannot be applied to the column in the select lists of a scalar-fullselect expression if the result of scalar-fullselect expression is derived from any set operations. That is, the application of column masks supports the scalar-subselect expression only.
3. If the subselect contains a GROUP BY clause, the column mask cannot be applied to a column in the corresponding select list if none of the following conditions is satisfied:
   - The column must identify a column-name in the GROUP BY clause and the column must not be referenced in an expression in the GROUP BY clause. Furthermore, its column mask definition must satisfy the following condition:
     - any columns that are referenced in the column mask definition that come from the same table as the column to which the column mask is applied must identify a *column-name* in the GROUP BY clause
     - the column mask must not be referenced in an expression in the GROUP BY clause

- The column must be specified in an aggregate function and its column mask definition must satisfy the following conditions:
  - The column mask definition must not reference a scalar-fullselect
  - The column mask definition must not reference an aggregate function

4. If the subselect contains a GROUP BY clause, and a column in the corresponding select list maps directly or indirectly to a column name or an expression in a materialized table expression or view, the column in the subselect where the GROUP BY is specified must be specified under an aggregate function.
5. If the subselect does not contain a GROUP BY clause, and a column in the corresponding select list is specified in an aggregate function, the column mask cannot be applied if the column mask definition references:
   - a scalar-fullselect
   - an aggregate function
6. If the FROM clause in a subselect references a recursive common table expression, and if the result of the recursive common table expression is used to derive the final result table, the column mask cannot be applied to a column that is referenced in the fullselect of the recursive common table expression.
7. If the FROM clause in a subselect contains a data-change-table-reference, and if an INCLUDE clause is specified as part of the SQL data change statement, the column mask cannot be applied to the columns that are used to derive the values for these additional columns in the outermost select list.
8. If the FROM clause in a subselect references an external table user-defined function or an inline SQL table user-defined function, and if the result of the function is used to derive the final result table, the column mask cannot be applied to the column that is an argument of the function.
9. If an OLAP specification is referenced in a select list that derives the final result table, the column mask cannot be applied to the column that is referenced in the partitioning expression or the sort key expression of the OLAP specification.
10. If a user-defined function is defined with the NOT SECURED option, the argument of the function must not reference a column for which a column mask is enabled and column access control is activated for its table. This rule applies to user-defined functions that are referenced anywhere in the statement.

To avoid the above error situations, one of the following actions must be taken:

- modify or remove the above contexts from the statement
- disable the column mask
- drop the column mask, modify the definition, and recreate the column mask
- deactivate column access control for the table

In other situations, if the statement contains a SELECT DISTINCT, and a column mask is applied to a column that directly or indirectly derives the result of SELECT DISTINCT, the statement might return a result that is not deterministic. The following examples illustrate when such results might be returned:

1. If the column mask definition references other columns from the same table as the column to which the column mask is applied, the result of SELECT DISTINCT can not be deterministic.

2. If the column is referenced in the argument of built-in scalar functions (such as COALESCE, IFNULL, NULLIF, MAX, MIN, LOCATE), the result of SELECT DISTINCT might not be deterministic.
3. If the column is referenced in the argument of an aggregation function, the result of SELECT DISTINCT might not be deterministic.
4. If the column is embedded in an expression and the expression contains a function that is not deterministic or has an external action, the result of SELECT DISTINCT might not be deterministic.

If the column is not nullable, most likely its column mask definition will not consider a null value for the column. After column access control is activated for the target table, if the target table is the null-padded table in an outer join operation, the column value in the final result table might be a null.

For INSERT, UPDATE, and MERGE, when a column is referenced while deriving the values of a new row, if that column has an enabled column mask, the masked value is used to derive the new values. If the object table also has column access control activated, the column mask that is applied to derive the new values must return the column itself, not a constant or an expression. If the column mask does not mask the column to itself, the new value cannot be used for insert or update and an error is returned. The rules that are used to apply column masks in order to derive the new values follow the same rules described above for the final result table of a query. See the data change statements for how the column masks are used to affect the insertability and updatability.

Column mask can be applied only to a base table column. If a materialized table expression, materialized view, or common table expression column is involved in the final result table, the above error situations can occur inside the materialized table expression, materialized view, or common table expression definition.

Column access control does not affect the XMLTABLE built-in function. If the input to the XMLTABLE function is a column with a column mask, the column mask is not applied.

Column access control remains activated until the DEACTIVATE COLUMN ACCESS CONTROL clause is used to stop enforcing it.

Stop enforcing row or column access control: The DEACTIVATE ROW ACCESS CONTROL clause is used to stop enforcing row access control for a table. The default row permission is dropped. Thereafter, when the table is referenced in a data manipulation statement, explicitly created row permissions are not applied. The table is accessible based on the granted privileges.

The DEACTIVATE COLUMN ACCESS CONTROL clause is used to stop enforcing column access control for a table. Thereafter, when the table is referenced in a data manipulation statement, the column masks are not applied. The unmasked column values are used for the final result table. The explicitly created row permissions or column masks, if any, remain but have no effect.

Secure triggers for row and column access control: Triggers are used for database integrity, and as such a balance between row and column access control (security) and database integrity is needed. Enabled row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row and column access control enforced for the triggering table is also ignored for any transition variables or transition tables referenced in the trigger body. To ensure

there is no security concern for SQL statements in the trigger action to access sensitive data in transition variables and transition tables, the trigger must be created or altered with the SECURED option. If a trigger is not secure, row and column access control cannot be enforced for the triggering table.

Secure user-defined functions for row and column access control: If a row permission or column mask definition references a user-defined function, the function must be altered with the SECURED option because the sensitive data might be passed as arguments to the function.

DB2 considers the SECURED option an assertion that declares the user has established a change control audit procedure for all changes to the user-defined function. It is assumed that such a control audit procedure is in place for all versions of the user-defined function, and that all subsequent ALTER FUNCTION statements or changes to external programs are being reviewed by this audit process.

Database operations where row and column access control is not applicable: Row and column access control must not compromise database integrity. Columns involved in primary keys, unique keys, indexes, check constraints, and referential integrity must not be subject to row and column access control. Column masks can be defined for those columns but they are not applied during the process of key building or constraint or RI enforcement.

Read-only cursors and read-only views: The rules that are used to determine a read-only cursor or a read-only view remain unaffected by row and column access control. The effect of application of enabled column masks is not known until run time. Therefore, the data change operation on a writable cursor or a writable view could still fail at run time.

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 1000.

```
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
  ADD CONSTRAINT DEPTQUIP
    FOREIGN KEY (EQUIP_OWNER)
      REFERENCES DEPARTMENT
      ON DELETE SET NULL
```

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 $16,000.

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT REVENUE
  CHECK (SALARY + COMM > 16000)
```

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 from CHAR(4) to accept up to 20 characters for a phone number. Since assignment rules are used to assign the CHAR(4) values to VARCHAR(20), any trailing blanks in the existing values for the column would continue to be part of the value. Use an UPDATE statement after the ALTER to remove these trailing blanks.

```
ALTER TABLE EMPLOYEE
  ALTER COLUMN PHONENO SET DATA TYPE VARCHAR (20)

UPDATE TABLE EMPLOYEE
  SET PHONENO = RTRIM(PHONENO, 1)
  WHERE RIGHT(PHONENO, 1) = ' '
```

Example 7: Alter the base table TRANSCOUNT to a materialized query table. The result of the *fullselect* 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 MATERIAILZED QUERY
  (SELECT ACCTID, LOCID, YEAR, COUNT(*) AS CNT
     FROM TRANS
     GROUP BY ACCTID, LOCID, YEAR )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER
```

Examples for column access control

Example 1: Based on the data in the CUSTOMER table, the SELECT DISTINCT statement returns one row with the SALARY value 100,000. A column mask, SALARY_MASK, is created to mask the salary value. After column access control is activated for the CUSTOMER table, the column mask is applied to SALARY column. A user with the 'MGR' authorization ID issues a SELECT DISTINCT statement. The SELECT DISTINCT statement still returns one row because the

removal of duplicates is based on the unmasked value of the SALARY column, but the value that is returned in that row is based on the masked SALARY value, which can be either 125,000 or 110,000.

The table CUSTOMER contains:

SALARY	COMMISSION	EMPID
100,000	25,000	123456
100,000	10,000	654321

```
CREATE MASK SALARY_MASK ON CUSTOMER
   FOR COLUMN SALARY RETURN
      CASE WHEN(SESSION_USER = 'MGR')
              THEN SALARY + COMMISSION
           ELSE SALARY
      END
   ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT DISTINCT SALARY FROM CUSTOMER;
```

Example 2: Based on the data in T1 and T2 tables, the SELECT DISTINCT statement using the COALESCE function returns one row with the T1.C1 value of 1. A column mask, C1_MASK, is created to mask the value of T1.C1. After column access control is activated for table T1, the column mask is applied to column C1 of table T1. A user with the 'EMP' authorization ID issues a SELECT DISTINCT statement. The SELECT DISTINCT statement still returns one row because the removal of duplicates is based on the unmasked value of T1.C1 from the COALESCE function, but the value that is returned in that row is based on the masked value of T1.C1 from the COALESCE function. The returned value can be either 2 or 3.

```
INSERT INTO T1(C1) VALUES(1);
INSERT INTO T1(C1) VALUES(1);

INSERT INTO T2(C1) VALUES(2);
INSERT INTO T2(C1) VALUES(3);

CREATE MASK C1_MASK ON T1
   FOR COLUMN C1 RETURN
      CASE WHEN(SESSION_USER = 'EMP')
         THEN NULL
         ELSE C1
      END
   ENABLE;

COMMIT;

ALTER TABLE T1
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT DISTINCT COALESCE(T1.C1, T2.C1) FROM T1, T2;
```

Example 3: Based on the data in the CUSTOMER table, the maximum income is the same in the states CA and IL, 50,000, thus, the SELECT DISTINCT statement returns one row. A column mask, INCOME_MASK, is created to mask the income value. After column access control is activated for the CUSTOMER table, the column mask is applied to the INCOME column before the MAX aggregate function is evaluated. However, the INCOME_ MASK column mask, masks the income value of 0 as 100,000 in state IL. As a result, the maximum income becomes 100,000 for state IL, but the maximum income is still 50,000 for state CA. X.B is used in a predicate in the SELECT DISTINCT statement, therefore, the original INCOME values and the original results of the MAX(INCOME) function must be preserved. So the SELECT DISTINCT statement still returns one row, but the value in that row might not be deterministic, that is, the value might be 50,000 from the 'CA' row or might be 100,000 from the 'IL' row.

The CUSTOMER table contains:

STATE	INCOME
CA	40,000
CA	50,000
IL	0
IL	10,000
IL	50,000

```
CREATE MASK INCOME_MASK ON CUSTOMER
   FOR COLUMN INCOME RETURN
      CASE WHEN(INCOME = 0)
         THEN 100000
         ELSE INCOME
      END
   ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT DISTINCT B FROM
   (SELECT STATE, MAX(INCOME) FROM CUSTOMER
      GROUP BY STATE)
      X(A, B)
   WHERE B > 10000;
```

Example 4: The expression INCOME + RAND() is not deterministic because the RAND function is not deterministic. Based on the data in the CUSTOMER table, the SELECT DISTINCT statement will, most likely, return two distinct rows. However, it could return only one row. A column mask, INCOME_MASK, is created to mask the income value. After column access control is activated for the CUSTOMER table, the column mask is applied to the INCOME column, which causes the masked value for both rows to be the same. Because the RAND function is not deterministic, the SELECT DISTINCT statement will, most likely, still return two distinct rows, but it could return only one row. The uncertainty caused by the RAND function causes the result of the SELECT DISTINCT statement to not be deterministic.

The CUSTOMER table contains:

STATE	INCOME
CA	40,000
CA	50,000

```
CREATE MASK INCOME_MASK ON CUSTOMER
   FOR COLUMN INCOME RETURN
      CASE WHEN(INCOME = 40,000)
         THEN 50000
         ELSE INCOME
      END
   ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT DISTINCT A FROM
   (SELECT INCOME + RAND() FROM CUSTOMER)
      X(A)
   WHERE A > 10000;
```

Example 5: A column mask, STATE_MASK, is created for the STATE column of the CUSTOMER table to return a value that shows the city name with the state if the city is SJ, SFO, or OKLD. Otherwise the city is not returned, just the state. After column access control is activated for the CUSTOMER table, a SELECT statement which groups results using the STATE column is issued. However, because the CITY column that is referenced in the STATE_MASK column mask is not a grouping column, an error is returned to signify that the STATE_MASK column mask is not appropriate for this statement.

The CUSTOMER table contains:

STATE	CITY	INCOME
CA	SJ	40,000
CA	SC	30,000
CA	SB	60,000
CA	SFO	80,000
CA	OKLD	50,000
CA	SJ	70,000
NY	NY	50,000

```
CREATE MASK STATE_MASK ON CUSTOMER
   FOR COLUMN STATE RETURN
      CASE WHEN(CITY = 'SJ')
          THEN CITY||', '||STATE
        WHEN(CITY = 'SFO')
          THEN CITY||', '||STATE
        WHEN(CITY = 'OKLD')
          THEN CITY||', '||STATE
        ELSE STATE
      END
   ENABLE;

COMMIT;
```

```
ALTER TABLE CUSTOMER
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT STATE, AVG(INCOME) FROM CUSTOMER
   GROUP BY STATE
   HAVING STATE = 'CA';
```

ALTER TRIGGER

The ALTER TRIGGER statement changes the description of the 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 authorization ID of the statement must have security administrator authority.

Syntax

Description

trigger-name

Identifies the trigger to be altered. The *trigger-name* must identify a trigger that exists at the current server.

SECURED or NOT SECURED

Specifies whether the trigger is considered secure for row and column access control. If row or column access control is active for the triggering table, altering the trigger from SECURED to NOT SECURED will return an error.

SECURED

Specifies that the trigger is considered secure for row access control and column access control.

SECURED must be specified for a trigger if row or column access control is active for the triggering table. SECURED must also be specified for a trigger that is created for a view and row or column access control is active for one or more of the underlying tables in the view definition.

NOT SECURED

Specifies that the trigger is considered not secure for row access control and column access control.

NOT SECURED must not be specified for a trigger if row or column access control is active for the triggering table. NOT SECURED must also not be specified for a trigger that is created for a view and row or column access control is active for one or more of the underlying tables in the view definition.

Notes

Altering a trigger from NOT SECURED to SECURED: The trigger is considered secure after the ALTER TRIGGER statement is executed. DB2 treats the SECURED attribute as an assertion that declares that the user has established an audit procedure for all activities in the trigger body. If a secure trigger references user-defined functions, DB2 assumes those functions are secure without validation. If those functions can access sensitive data, a user with security administrator

authority needs to ensure that those functions are allowed to access that data and that an audit procedure is in place for those functions, and that all subsequent ALTER FUNCTION statements are being reviewed through this audit process.

Transition variable values and row and column access control: Row and column access control is not enforced for transition variables and transition tables. If row or column access control is enforced for the triggering table, row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row and column access control enforced for the triggering table is also ignored for transition variables and transition tables that are referenced in the trigger body or are passed as arguments to user-defined functions invoked within the trigger body. To ensure there are no security concerns for SQL statements accessing sensitive data in transition variables and transition tables, the trigger must be created with the SECURED option. If the trigger is not secure, row access control and column access control cannot be activated for the triggering table.

Examples

Example 1: Change the definition of trigger TRIGGER1 to secured:

```
ALTER TRIGGER TRIGGER1
  SECURED
```

Example 2: Change the definition of trigger TRIGGER1 to not secured:

```
ALTER TRIGGER TRIGGER1
  NOT SECURED
```

ASSOCIATE LOCATORS

The ASSOCIATE LOCATORS statement gets the result set locator value for each result set returned by a procedure.

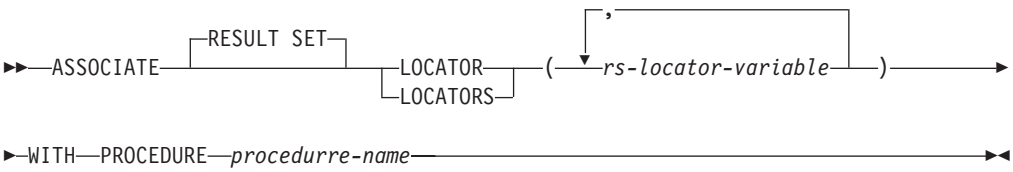
Invocation

This statement can only be embedded in an SQL procedure. It is not an executable statement and cannot be dynamically prepared.

Authorization

None required.

Syntax



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*
Identifies the procedure that returned one or more result sets. The *procedure-name* must identify a procedure that exists at the current server. 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.

The procedure name in the ASSOCIATE LOCATORS statement must be specified the same way that it was specified on the CALL statement. For example, if a two-part procedure name was specified on the CALL statement, you must specify a two-part procedure name in the ASSOCIATE LOCATORS statement.

Notes

Assignment of locator values. If the ASSOCIATE LOCATORS statement specifies multiple locator variables, locator values are assigned to the locator variables in the order that the associated cursors are opened at runtime.

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 opened again 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: If a procedure is called more than once from the same caller, only the most recent result sets are accessible.

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.

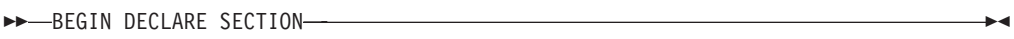
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java 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 783.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and must not be nested.

SQL statements must not be included within an SQL declare section, with the exception of INCLUDE statements that include host variable declarations.

Host variables referenced in SQL statements must be declared in an SQL declare section in all host languages, other than Java and REXX. Furthermore, the declaration of each variable must appear before the first reference to the variable. Host variables are declared without the use of these statements in Java, 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

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The authorization ID of the statement must have at least one of the following:

- The EXECUTE privilege on the procedure
- Ownership of the procedure
- Database administrator authority

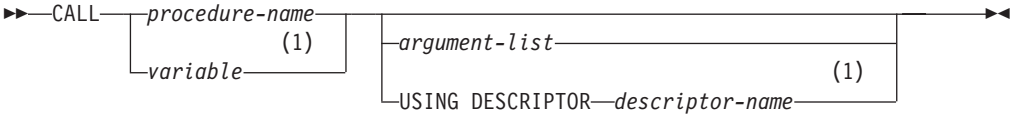
If a global variable is referenced as an IN or INOUT parameter, the privileges held by the authorization ID for the statement must include at least one of the following:

- The READ privilege on the global variable
- Database administrator authority

If a global variable is referenced as an OUT or INOUT parameter, the privileges held by the authorization ID for the statement must include at least one of the following:

- The WRITE privilege on the global variable
- Database administrator authority

Syntax



argument-list:

Notes:

1 DB2 for LUW requires use of a product-specific program preparation option.

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 CHAR or VARCHAR variable with a length attribute that is not greater than 254 bytes. It cannot be a global variable.
- It must not be followed by an indicator variable.
- The value within the variable must be left justified and must not contain any embedded blanks.
- If the variable is a fixed length string, the value within the variable must be padded on the right with blanks if its length is less than that of the variable.
- The value within the variable must be in uppercase characters 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 52.

The procedure definition at the current server determines the name of the external program, language, and calling convention of the procedure. See "CREATE PROCEDURE" on page 666 for more information.

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 CREATE PROCEDURE) as OUT or INOUT must be specified as a variable.

The number of arguments specified must be the same as the number of parameters of a procedure defined at the current server with the specified *procedure-name*.

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.

A variable cannot be a structure when used with the CALL statement.

expression
An *expression* of the type described in "Expressions" on page 130, 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.

NULL
Specifies a null value as an argument to the procedure.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of host variables that are passed as parameters to the procedure. If the procedure has no parameters, the SQLDA is ignored.

Before the CALL statement is processed, the user must set the following fields in the SQLDA (Note that the rules for REXX are different. For more information, see Chapter 18, "Coding SQL statements in REXX applications," on page 1117):

- 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 on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see Chapter 12, "SQLDA (SQL descriptor area)," on page 985.

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 for the procedure. The *n*th variable described by the SQLDA corresponds to the *n*th parameter in the procedure.

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 arguments is assigned (using storage assignment) to the corresponding parameter of 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) to the corresponding argument of the CALL statement defined as OUT or INOUT. If an error is returned by the procedure, OUT arguments are undefined and INOUT arguments are unchanged. For details on the assignment rules, see "Assignments and comparisons" on page 77.

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.[111]

Result sets from procedures: Any cursors specified using the WITH RETURN clause that the procedure leaves open when it returns identifies a result set. In a procedure written in Java, all cursors are implicitly defined WITH RETURN TO CALLER.

Results sets are returned only when the procedure is called from CLI, JDBC, or SQLJ. If the procedure was invoked from CLI or Java, and more than one cursor is left open, the result sets can only be processed in the order in which the cursors were opened. Only unread rows are available to be fetched. For example, if the

111. Product-specific options exist that may extend the scope of cursors and prepared statements.

result set of a cursor has 500 rows, and 150 of those rows have been read by the procedure at the time the procedure is terminated, then rows 151 through 500 will be returned to the procedure.

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 can issue a CALL statement. When a procedure calls another procedure, the call is considered to be nested. If a nested procedure returns a result set, the result set is available only to the immediate caller of the nested procedure.

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'.

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 749 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: At the end of a unit of work, all open cursors declared without the WITH HOLD option that belong to an application process are implicitly closed.

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 584.

Examples

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 a comment in the catalog descriptions of an 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

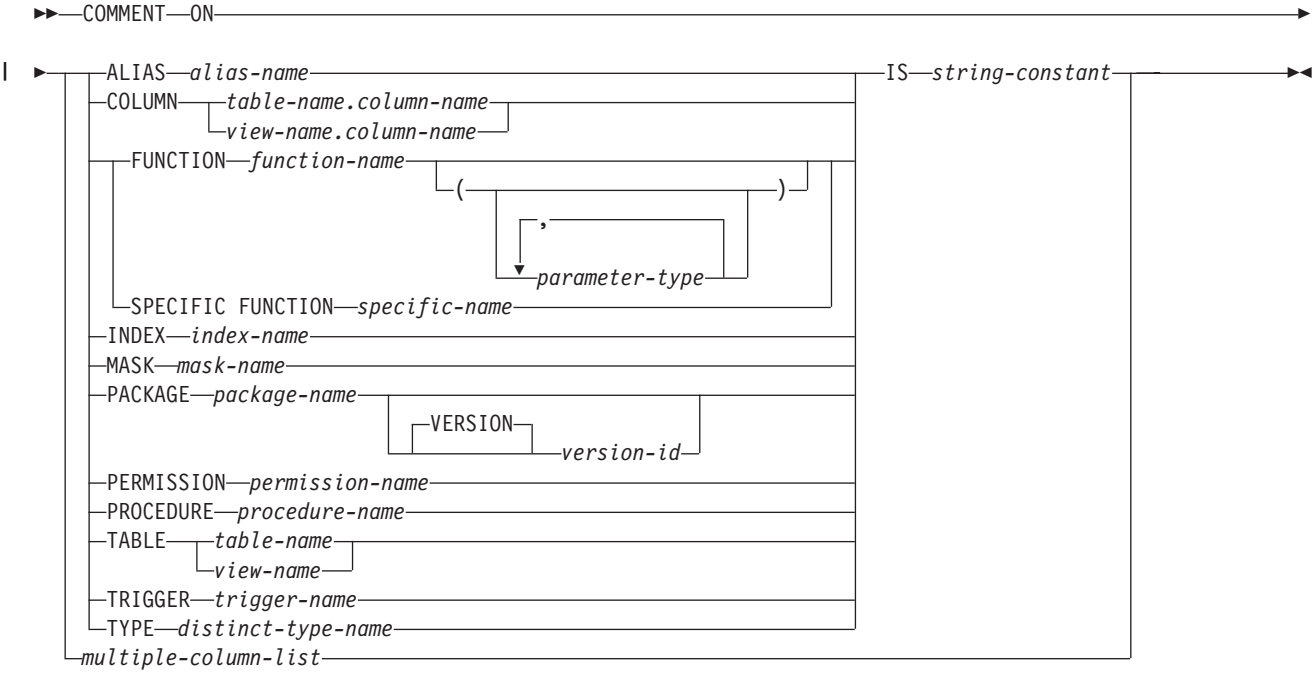
To comment on a mask or permission:

- The authorization ID of the statement must have Security administrator authority.

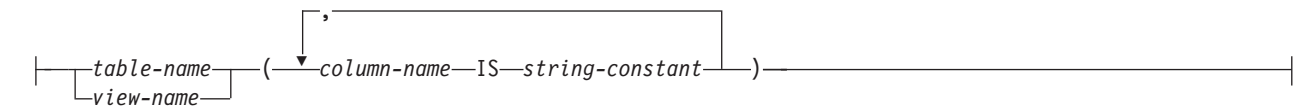
For other objects, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the object
- Database administrator authority

Syntax



multiple-column-list:

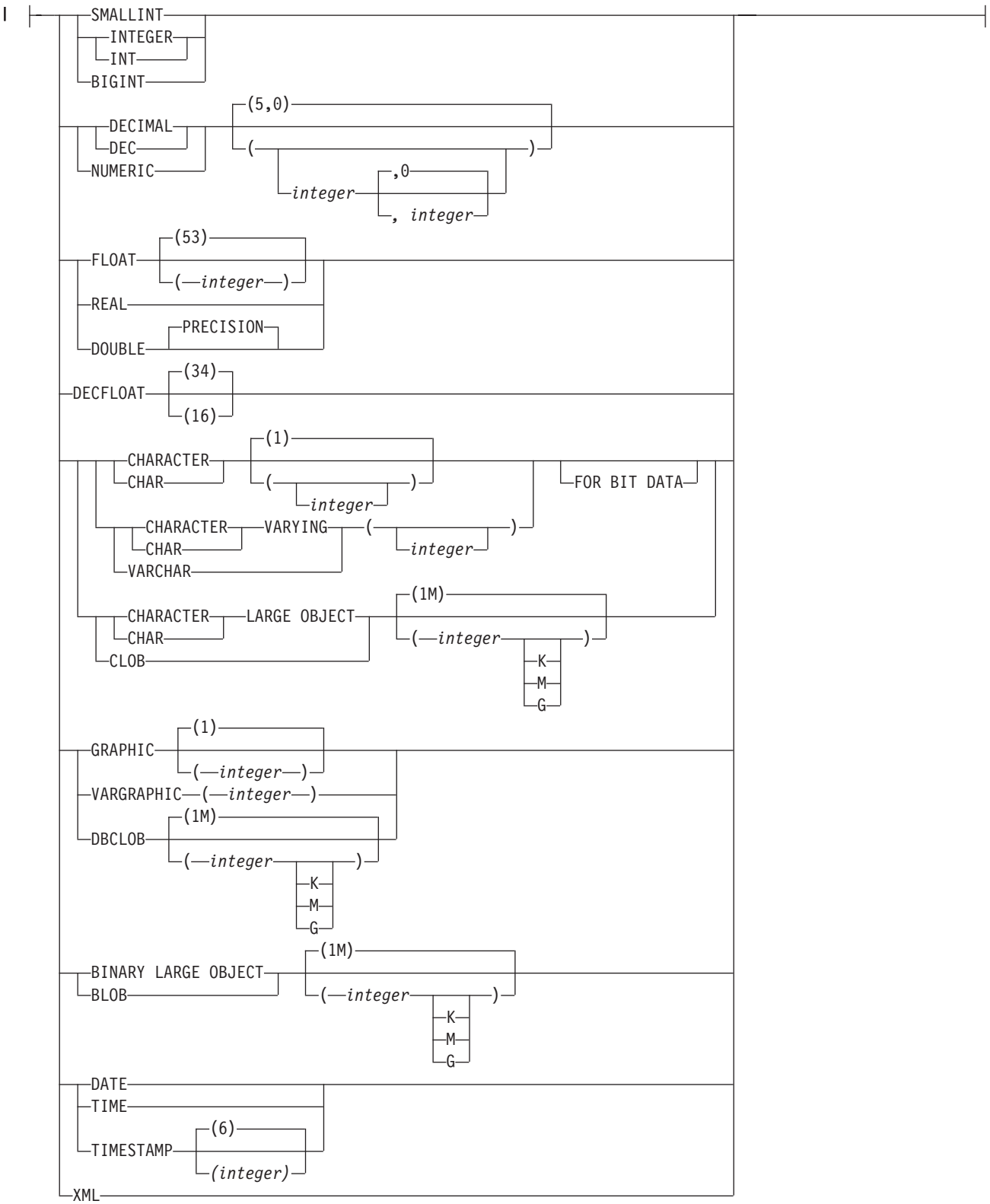


parameter-type:

data-type:

COMMENT

built-in-type:



Description

ALIAS *alias-name*
Identifies the alias to which the comment applies. *alias-name* must identify an alias that exists at the current server.

COLUMN
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.

FUNCTION or SPECIFIC FUNCTION
Identifies the function to which the comment applies. The function must exist at the current server and it must be a function that was defined with the CREATE FUNCTION statement or a cast function that was generated by a CREATE TYPE statement. The particular 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 to which the comment applies. Synonyms for data types are considered a match. The rules for function resolution (and the SQL path) are not used.

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 indicates that the database manager ignores the attribute when determining whether the data types match. For example, DECIMAL() will be considered a match for a parameter of a function defined with a data type of DECIMAL(7,2). However, FLOAT cannot be specified with empty parentheses 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.

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.

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. *index-name* must identify an index that exists at the current server.

MASK *mask-name*
Identifies the mask to which the comment applies. The *mask-name* must identify a mask that exists at the current server.

PACKAGE *package-name*
Identifies the package to which the comment applies. *package-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.

PERMISSION *permission-name*
Identifies the permission to which the comment applies. The *permission-name* must identify a permission that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure to which the comment applies. *procedure-name* must identify a procedure that exists at the current server.

TABLE *table-name* **or** *view-name*
Identifies the table or view to which the comment applies. The name must identify a table or view that exists at the current server and must not identify a declared temporary table.

TRIGGER *trigger-name*
Identifies the trigger to which the comment applies. *trigger-name* must identify a trigger that exists at the current server.

TYPE *distinct-type-name*
Identifies the distinct type to which the comment applies. *distinct-type-name* must identify a distinct type that exists at the current server.

IS
Introduces the comment to be added or replaced.

string-constant
Can be any character string constant of up to 254 characters.

multiple-column-list
To comment on more than one column in a table or view with a single COMMENT statement, specify the table or view name, followed by a list in parentheses of the form:

```
(column-name IS string-constant,
 column-name IS string-constant, ... )
```

Each column name must not be qualified, and must identify a column of the specified table or view that exists at the current server.

Examples

Example 1: Add a comment for the EMPLOYEE table.

```
COMMENT ON TABLE EMPLOYEE
  IS 'Reflects first quarter 2000 reorganization'
```

Example 2: Add a comment for the EMP_VIEW1 view.

```
COMMENT ON TABLE EMP_VIEW1
  IS 'View of the EMPLOYEE table without salary information'
```

Example 3: Add a comment for the EDLEVEL column of the EMPLOYEE table.

```
COMMENT ON COLUMN EMPLOYEE.EDLEVEL
  IS 'highest grade level passed in school'
```

Example 4: Add comments for two different columns of the DEPARTMENT table.

```
COMMENT ON DEPARTMENT
  (MGRNO IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',
   ADMRDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT')
```

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.

Authorization

None required.

Syntax

Description

The COMMIT statement ends the unit of work in which it is executed. It commits all changes made by SQL schema statements and SQL data change statements during the unit of work. For more information see Chapter 7, “Statements,” on page 519.

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.

Effect of commit: Commit causes the following to occur:

- Connections in the release-pending state are ended. Some products provide options that cause remote connections in the held state to be 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 are closed.
- All LOB locators 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.
- For DB2 for z/OS, prepared statements are destroyed, except those statements required for the cursors that were not closed.

Other transaction environments: SQL COMMIT may not be available in other transaction environments, such as IMS™ and CICS®. To do a commit operation in these environments, SQL programs must use the call prescribed by their transaction manager.

Examples

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;
      return;
    SQLERR:
      ...
      EXEC SQL WHENEVER SQLERROR CONTINUE; /* continue if error on rollback */
      EXEC SQL ROLLBACK;
      return;
  }
```

CONNECT (type 1)

The CONNECT (type 1) statement connects an application process to the identified application server and establishes the rules for remote unit of work. This server is then the current server for the process. Differences between this statement and the CONNECT (Type 2) statement are described in "CONNECT (type 1) and CONNECT (type 2) differences" on page 979. Refer to "Application-directed distributed unit of work" on page 39 for more information about connection states.

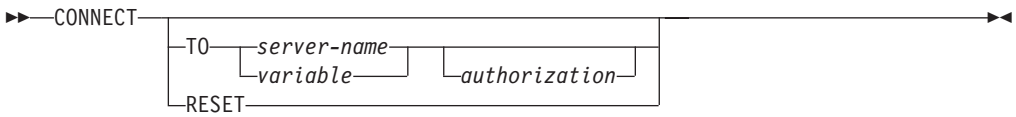
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

The authorization ID of the statement must be authorized to connect to the identified application server. The authorization check is performed by that server.
G The authorization required is product-specific.

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. If a variable is specified:

- It must be a CHAR or VARCHAR variable with a length attribute that is not
G greater than 18. It must not be a global variable. In DB2 for z/OS, the
G maximum length is 16. In DB2 for LUW, the maximum length is 8.
- 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 and the application process must be in the

G connectable state. (See "Notes" for information about connection states.) In
G DB2 for LUW, the server name is a database alias name identifying the
G application server.

USER *variable*
Identifies the authorization name that will be used to connect to the application server. The *variable* must satisfy the following:

- It must be a CHAR or VARCHAR variable with a length attribute that is not
| greater than 8. It must not be a global variable. See Table 63 on page 967 for more information.
- 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 for 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 authorization name must not contain lowercase characters.

G For DB2 for z/OS, authorization may not be specified when the connection
G type is IMS or CICS.

USING *variable*
Identifies the password that will be used to connect to the application server. The *variable* must satisfy the following:

- It must be a CHAR or VARCHAR variable with a length attribute that is not
| greater than 8. It must not be a global variable. See Table 63 on page 967 for more information.
- 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 the length 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.

G For DB2 for LUW, CONNECT RESET only disconnects the application process
G from the current server. If implicit connect is available, the application process
G remains unconnected until an SQL statement is issued.

CONNECT with no operand
This form of the CONNECT statement returns information about the current server and has no effect on connection states. The information is returned in the SQLERRP field of the SQLCA as described below.

Notes

Successful connection: If the CONNECT statement (excluding the CONNECT with no operand form) is successful:

- All open cursors are closed, all prepared statements are destroyed, all locators are freed, and all locks are released from the previous application server.
- The application process is disconnected from its previous application server, 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 SQLERRP field of the SQLCA. The format below applies if the application server is a DB2 product. The information has the form *pppvvrrm*, where:
  - *ppp* is:
    DSN for DB2 for z/OS
    QSQ for DB2 for i
    SQL for DB2 for LUW
  - *vv* is a two-digit version identifier such as '09'.
  - *rr* is a two-digit release identifier such as '01'.
  - *m* is a one-digit modification level such as '0'.

  For example, if the server is Version 9 of DB2 for z/OS, the value would be 'DSN09010'.
- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. The contents are product-specific.

Unsuccessful connection: If the CONNECT statement is unsuccessful, 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 identifies the product. For example, if the application requester is DB2 for LUW, the first three characters are 'SQL'.

If the CONNECT statement is unsuccessful because the application process is not in the connectable state, the connection state of the application process is unchanged.

If the CONNECT statement is unsuccessful because the *server-name* is not listed in the local directory, the connection state of the application process is product-specific.

If the CONNECT statement is unsuccessful for any other reason, the application process is placed in the unconnected state, all open cursors are closed, all prepared statements are destroyed, and any held resources are released.

For a description of connection states, see "Remote unit of work connection management" on page 37. See the description of the CONNECT statement in your product's SQL reference for further information.

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 3 character product identifier of the application server to the variable PRODUCT (CHAR(3)).

```
void main ()
  {
    char product[4] = " ";
    EXEC SQL BEGIN DECLARE SECTION;
    char APP_SERVER[19];
    char username[129];
    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) )
    { strncpy(product,sqlca.sqlerrp,3); }
  ...
  return;
}
```

CONNECT (type 2)

The CONNECT (type 2) statement connects the application process to the identified application server and establishes the rules for application-directed distributed unit of work. This server is then the current server for the process. Differences between this statement and the CONNECT (Type 1) statement are described in "CONNECT (type 1) and CONNECT (type 2) differences" on page 979. Refer to "Application-directed distributed unit of work" on page 39 for more information about connection states.

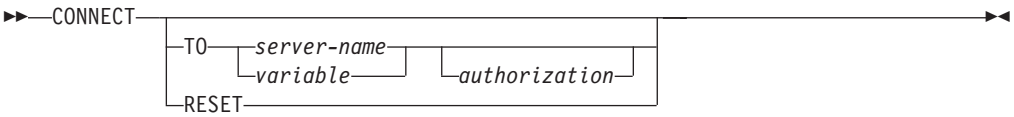
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

The authorization ID of the statement must be authorized to connect to the identified application server. The authorization check is performed by that server.
G The authorization required is product-specific.

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. If a variable is specified:

- It must be a CHAR or VARCHAR variable with a length attribute that is not
G greater than 18. It must not be a global variable. In DB2 for z/OS, the
G maximum length is 16. In DB2 for LUW, the maximum length is 8.
- 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. The application process must not have an existing connection to S. [112]

USER *variable*
Identifies the authorization name that will be used to connect to the application server. The *variable* must satisfy the following:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 8. It must not be a global variable. See Table 63 on page 967 for more information.
- 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 for 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 authorization name must not contain lowercase characters.

USING *variable*
Identifies the password that will be used to connect to the application server. If the *variable* is specified:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 8. It must not be a global variable. See Table 63 on page 967 for more information.
- 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 the length of the variable, it must be padded on the right with blanks.
- The value of the password must not contain lowercase characters.

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. The information is returned in the SQLERRP field of the SQLCA as described below.

Notes

Successful connection: If the CONNECT statement (excluding the CONNECT with no operand form) is successful:

- A connection to application server S is created and placed in the current and held states. The previously current connection, if any, is placed in the dormant state.
- S is placed in the CURRENT SERVER special register.
- Information about application server S is placed in the SQLERRP field of the SQLCA. The format below applies if the application server is a DB2 product. The information has the form *pppvvrrm*, where:
  - *ppp* is:
    DSN for DB2 for z/OS
    QSQ for DB2 for i

112. In DB2 for z/OS, this rule is enforced only if the SQLRULES(STD) bind option is specified.

SQL for DB2 for LUW
  - *vv* is a two-digit version identifier such as '09'.
  - *rr* is a two-digit release identifier such as '01'.
  - *m* is a one-digit modification level such as '0'.

  For example, if the server is Version 9 of DB2 for z/OS, the value would be 'DSN09010'.
- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. The contents are product-specific.

Unsuccessful connection: If the CONNECT statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.

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)
```

CREATE ALIAS

The CREATE ALIAS statement defines an alias for a table or view.

Invocation

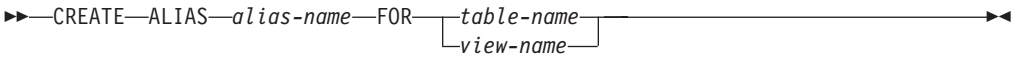
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema
- Database administrator authority.

Syntax



Description

alias-name
Names the alias. The name, including the implicit or explicit qualifier, must not be the same as an index, table, view or alias that already exists at the current server.

If the *alias-name* is qualified, the schema name must not be a system schema.

FOR *table-name* **or** *view-name*
Identifies the table or view at the current server for which *alias-name* is defined. An alias name must not be specified (an alias cannot refer to another alias).

An alias can be defined for an object that does not exist at the time of the definition. If it does not exist when the alias is created, a warning is returned. However, the referenced object must exist when a SQL statement containing the alias is used, otherwise an error is returned.

Examples

Example: Create an alias named CURRENT_PROJECTS for the PROJECT table.

```
CREATE ALIAS CURRENT_PROJECTS
  FOR PROJECT
```

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. For information on creating an external scalar function, see "CREATE FUNCTION (external scalar)" on page 610.
- **External table**

  The function is written in a programming language such as C, 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. For information on creating an external table function, see "CREATE FUNCTION (external table)" on page 622.
- **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 value, or the result of an aggregate function. For information on creating a sourced function, see "CREATE FUNCTION (sourced)" on page 632. The function inherits attributes of the underlying source function.
- **SQL scalar**

  The function is written exclusively in SQL statements and returns a scalar value. The body of an SQL scalar function is written in the SQL procedural language. The body of the function is defined at the current server along with various attributes of the function. For information on creating an SQL scalar function, see "CREATE FUNCTION (SQL scalar)" on page 640.

Notes

Choosing the schema and function name: If a qualified function name is specified, the *schema-name* must not be one of the system schemas (see "Schemas" on page 11). If *function-name* is not qualified, it is implicitly qualified with the default schema name.

The unqualified function name must not be the same as the name of a built-in data type, or any of the following, even they are specified as delimited identifiers:

```
ALL                HASHED_VALUE       STRIP              XMLPARSE
AND                IN                 SUBSTRING          XMLPI
ANY                IS                 TABLE              XMLROW
ARRAY_AGG          LIKE               THEN               XMLSERIALIZE
BETWEEN            MATCH              TRIM               XMLTEXT
BOOLEAN            NODENAME           TRUE               XMLVALIDATE
CASE               NODENUMBER         TYPE               XSLTRANSFORM
CAST               NOT                UNIQUE             =
CHECK              NULL               UNKNOWN            ¬=
DATAPARTITIONNAME  ONLY               WHEN               <
DATAPARTITIONNUM   OR                 XMLAGG             <=
DBPARTITIONNAME    OVERLAPS           XMLATTRIBUTES      ¬<
DBPARTITIONNUM     PARTITION          XMLCOMMENT         >
DISTINCT           POSITION           XMLCONCAT          >=
EXCEPT             RID                XMLDOCUMENT        ¬>
EXISTS             RRN                XMLELEMENT         !<
```

```
EXTRACT          SELECT           XMLFOREST        <>
FALSE            SIMILAR          XMLGROUP         !>
FOR              SOME             XMLNAMESPACES    !=
FROM
```

Defining the parameters: The input parameters for the function are specified as a list within parentheses.

The maximum number of parameters allowed in CREATE FUNCTION is 90. For more details on the limits on the number of parameters, see Chapter 9, "SQL limits," on page 959. DB2 for z/OS only uses the first 30 parameters to determine uniqueness.

A function can have no input parameters. In this case, an empty set of parentheses 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. For more information, see "Rules for result data types" on page 91. 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, use the following recommended data type names:
  - DOUBLE or REAL instead of FLOAT.
  - DECIMAL instead of NUMERIC.
- **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 data type or a distinct type that is based on a LOB 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 must not be specified for a sourced or SQL function.

  AS LOCATOR must not be specified if the function is defined with NO SQL.

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, or scale) 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. If the function has more than 30 parameters, DB2 for z/OS only considers the first 30 parameters to determine whether the function is unique.

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 the SPECIFIC clause is not specified, a specific name is generated.

Extending or overriding a built-in function: Giving a user-defined external 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. For information on creating sourced functions, see "CREATE FUNCTION (sourced)" on page 632. 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, it might be useful to use different rules for rounding than the built-in ROUND function. 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. This can occur with dynamic SQL statements, or when static SQL applications are rebound. For more information on when static statements are rebound, see "Packages and access plans" on page 18.

  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.

Creating a secure function: The database manager treats the SECURED attribute as an assertion that declares that the user has established an audit procedure for all changes to the user-defined function. The database manager assumes that all subsequent ALTER FUNCTION statements are being reviewed through this audit process.

Invoking other user-defined functions in a secure function: When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, a user with Security administrator authority needs to ensure that those functions are allowed to access that data and should ensure that a change control audit procedure has been established for all changes to those functions.

CREATE FUNCTION (external scalar)

The 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:

- The privilege to create in the schema
- Database administrator authority.

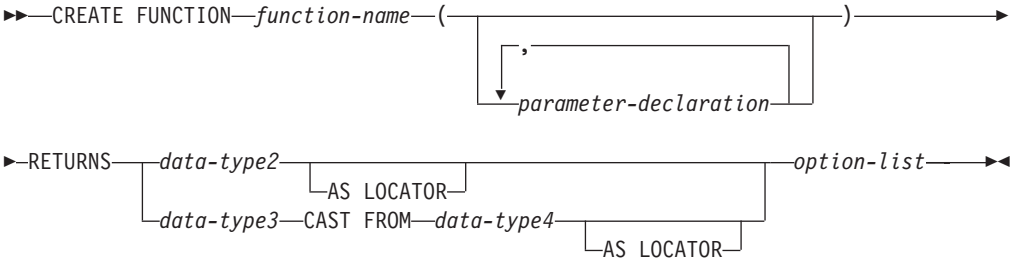
If the SECURED option is specified:

- The authorization ID of the statement must have Security administrator authority.

For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

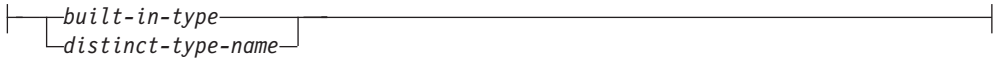
- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Database administrator authority.

Syntax

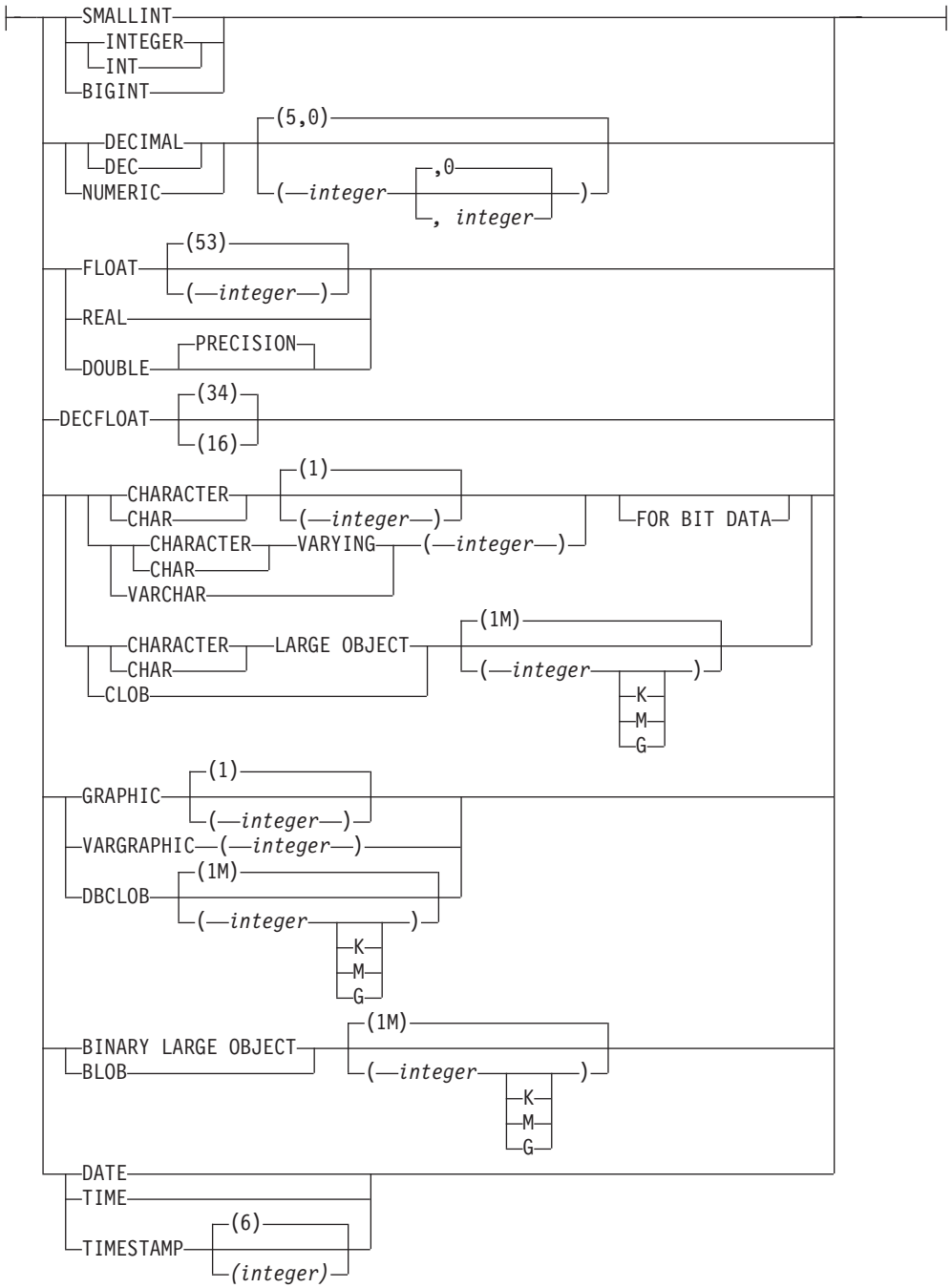


parameter-declaration:

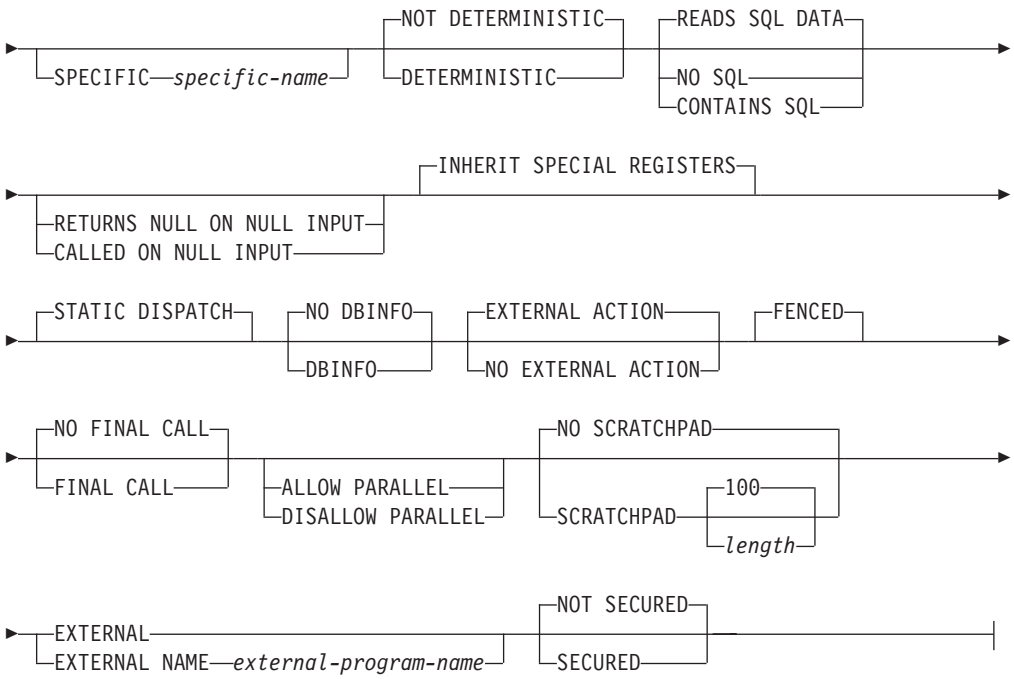
data-type1, data-type2, data-type3, data-type4:



built-in-type:



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

function-name
Names the user-defined function. If *function-name* is not qualified, it is implicitly qualified with the default schema name. The schema must be a valid schema name for functions, and the unqualified function name must not be any of the reserved function names. The same name can be used for more than one function in the same schema if the function signature of each function is unique. For more information about naming functions, see Choosing the schema and function name and Determining the uniqueness of functions in a schema.

(*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 688. 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 "Attributes of the arguments of a routine program" on page 1136. Built-in data type specifications can be specified if they correspond to the language that is used to write the user-defined function.

For parameters with a LOB data type, AS LOCATOR must not be specified when LANGUAGE JAVA is specified.

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 734.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

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 data type or a distinct type that is based on a LOB data type. AS LOCATOR must not be specified if the function is defined with NO SQL. Specify AS LOCATOR only when LANGUAGE JAVA is not specified.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

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 of the result.

The same considerations that apply to the data type and nullability of input parameters, as described under data-type1, apply to the data type of the result of the function.

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 that is based on a LOB data type. AS LOCATOR must not be specified if the function is defined with NO SQL. Specify AS LOCATOR only when LANGUAGE JAVA is not 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 of the result 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(DECIMAL(15,0))
      RETURNS DECIMAL(15,0)
      CAST FROM DOUBLE
      ...
```

The value of *data-type4* must not be a distinct type, and it 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 74.

AS LOCATOR
Specifies that the external program returns a locator to the value rather than the value. The value that is represented by this locator is then cast to *data-type3*. Specify AS LOCATOR only if *data-type4* is a LOB data type or a distinct type that is based on a LOB data type. AS LOCATOR must not be specified if the function is defined with NO SQL.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

LANGUAGE
Specifies the language interface convention to which the body of the function is written. All programs must be designed to run in the server's environment.

C Specifies that the external function is written in C or C++. The database manager will invoke the function using the C language calling conventions.

JAVA
Specifies that the external function is written in Java. The database manager will invoke the 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.

PARAMETER STYLE
Specifies the conventions for passing parameters to and returning a value from the function. For more information, see "Parameter passing for external routines" on page 1127.

SQL
Specifies the parameter passing convention that supports passing null values both as input and for output. PARAMETER STYLE SQL must be specified when LANGUAGE C is also specified. The parameters that are passed between the invoking SQL statement and the function include the following, in the order listed:

- *n* parameters for the input parameters that are specified for the function
- A parameter for the result of the function
- *n* parameters for the indicator variables for the input parameters
- A parameter for the indicator variable for the result
- The SQLSTATE that is to be returned to the database manager
- The qualified name of the function
- The specific name of the function

- The SQL diagnostic string that is to be returned to the database manager

The function can also pass from zero to three additional parameters:

- The scratchpad, if SCRATCHPAD is specified
- The call type, if FINAL CALL is specified
- The DBINFO structure, if DBINFO is specified

JAVA
Specifies that the user-defined function uses a convention for passing parameters that conforms to the Java and ISO/IEC FCD 9075-13:2008, *Information technology - Database languages - SQL - Part 13: Java Routines and Types (SQL/JRT)* specifications. PARAMETER STYLE JAVA must be specified when LANGUAGE JAVA is also specified and cannot be specified with any other LANGUAGE value.

When PARAMETER STYLE JAVA is specified, do not specify SCRATCHPAD, FINAL CALL, or DBINFO.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a function.

NOT DETERMINISTIC or 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.

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. An example of a deterministic function is a function that calculates the square root of the input argument.

The database manager does not verify that the function program is consistent with the specification of NOT DETERMINISTIC or DETERMINISTIC.

READS SQL DATA, NO SQL, or CONTAINS 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 "SQL statement data access classification for routines" on page 974. 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.

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 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.

DBINFO
Specifies that an additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application run-time authorization ID, the schema name, and identification of the database manager that invoked the function. For details about the argument and its structure, see "Database information in external routines (DBINFO)" on page 1138 or the sqludf include file.

Do not specify DBINFO when LANGUAGE JAVA is specified.

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 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.

A function with external actions might return incorrect results if the function is executed by parallel tasks. For example, if the function sends a note for each initial 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 do not work correctly with parallelism.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

The database manager does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

FENCED
Specifies that the external function runs in an environment that is isolated from the database manager environment.

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.

Do not specify FINAL CALL when LANGUAGE JAVA is specified.

The types of calls are:

First call
Specifies that this call is 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 that this call is the last call to the function, which enables the function to free resources. A final call is not a normal 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 the step of closing the cursor.

If a commit operation occurs while a cursor that is 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 that is defined as WITH HOLD is open.

If a commit, or rollback causes the final call, the function cannot issue any SQL statements except for the CLOSE statement.

A function that is defined with final call might return incorrect results if the function is executed by parallel tasks. 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 take inappropriate actions when executed in parallel.

ALLOW PARALLEL or DISALLOW PARALLEL

For a single reference to the function, specifies whether parallelism can be used when the function is invoked. Although parallelism can be used for most scalar functions, some functions, such as those that depend on a single copy of the scratchpad, cannot be invoked with parallel tasks. Consider these characteristics when determining which clause to use:

- If all invocations of the function are completely independent from one another, specify ALLOW PARALLEL.
- If each invocation of the function updates the scratchpad, to provide values that are of interest to the next invocation, such as incrementing a counter, specify DISALLOW PARALLEL.
- If the scratchpad is used only so that some expensive initialization processing is performed a minimal number of times, specify ALLOW PARALLEL.

The default is DISALLOW PARALLEL if one or more of the following clauses are specified: NOT DETERMINISTIC, EXTERNAL ACTION, FINAL CALL, 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.

Other attributes of a function might restrict parallelism, for more information see the description of NOT DETERMINISTIC, EXTERNAL ACTION, SCRATCHPAD, and FINAL CALL.

DISALLOW PARALLEL

Specifies that the database manager must not use parallelism for the function.

NO SCRATCHPAD or SCRATCHPAD

Specifies whether the database manager provides a scratchpad for the function. A scratchpad provides an area for the function to save information from one invocation to the next. The default is NO SCRATCHPAD.

NO SCRATCHPAD

Specifies that a scratchpad is not allocated and passed to the function.

SCRATCHPAD *length*

Specifies that when the function is invoked for the first time, the database manager allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.
- The database manager initializes the scratchpad to all binary zeros (X'00').
- The scope of a scratchpad is the SQL statement. Each reference to the function in an SQL statement, has 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 by the SQL statement. Specify the DISALLOW PARALLEL clause for functions that will not work correctly with parallelism.

- The scratchpad is persistent. The database manager preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. The database manager initializes the scratchpads when it begins to execute an SQL statement. The database manager does not reset scratchpads when a correlated subquery begins to execute.
- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify FINAL CALL to ensure that the database manager calls the function one more time so that the function can free those system resources.

Each time that the function invoked, the database manager passes an additional argument to the function that contains the address of the scratchpad.

Do not specify SCRATCHPAD when LANGUAGE JAVA is specified.

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 the NAME clause is not specified, 'NAME *function-name*' is implicit. In this case, *function-name* must not be longer than 8 characters. For more information on the maximum length of an external program name, see Chapter 9, “SQL limits,” on page 959. The NAME clause is required for a LANGUAGE JAVA function because the default name is not valid for a Java function.

NAME *external-program-name*
Specifies the program that is to be executed when the function is invoked. The executable form of the external program need not exist when the CREATE FUNCTION statement is executed. However, it must exist at the current server when the function is invoked.

If a JAR is referenced, the JAR must exist when the function is created.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row access control and column access control.

NOT SECURED
Specifies that the function is considered not secure for row access control and column access control. This is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

A function must be defined as secure when it is referenced in a row permission or a column mask.

Notes

General considerations for defining user-defined functions: For general information on defining user-defined functions, see "CREATE FUNCTION" on page 606.

Owner privileges: The owner is authorized to execute the function (EXECUTE) and has the ability to grant these privileges to others. For more information, see "GRANT (function or procedure privileges)" on page 793. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

Language considerations: A C program must be written to run as a subroutine.

For information on creating the programs for a function, see Chapter 19, "Coding programs for use by external routines," on page 1127.

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
```

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.

```
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 database administrator 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 (CLOB(128K))
   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)

The CREATE FUNCTION (external table) statement defines an external table function at the current server. A user-defined external table function can be used in the FROM clause of a subselect, and it 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:

- The privilege to create in the schema
- Database administrator authority.

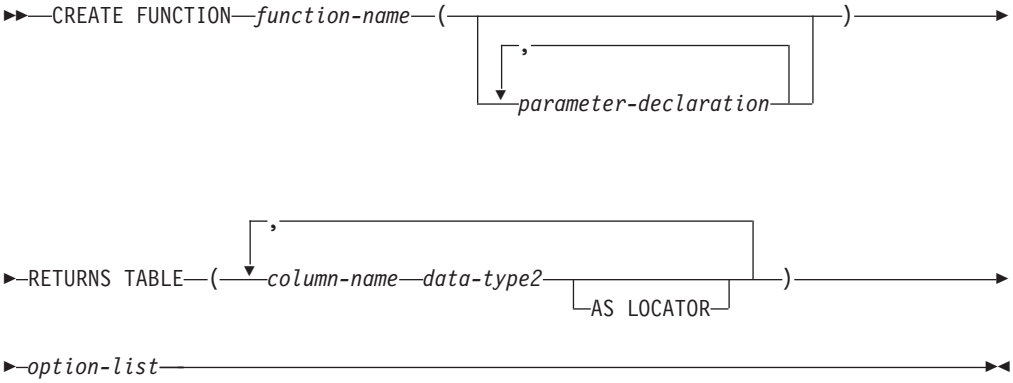
If the SECURED option is specified:

- The authorization ID of the statement must have Security administrator authority.

For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Database administrator authority.

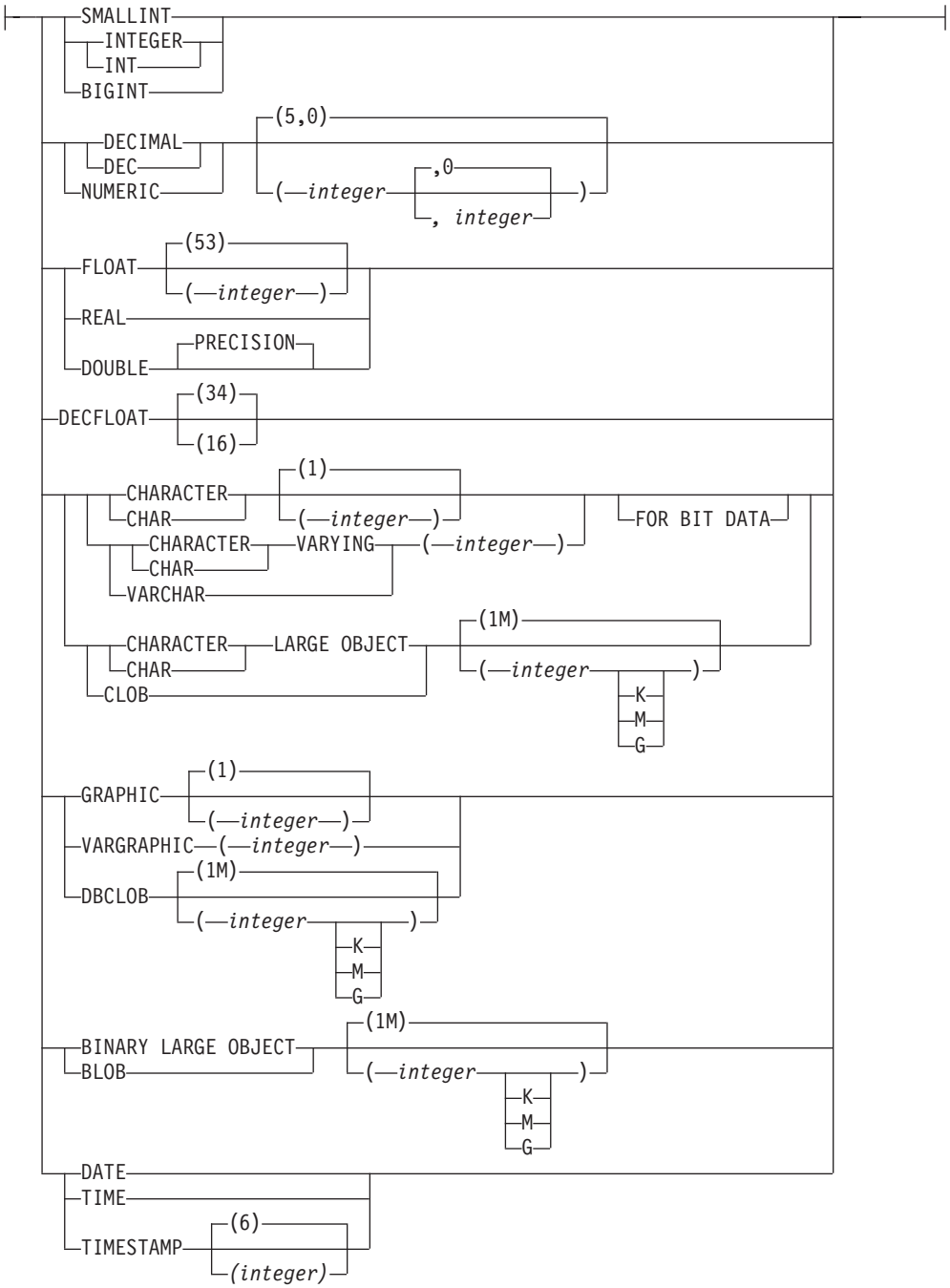
Syntax



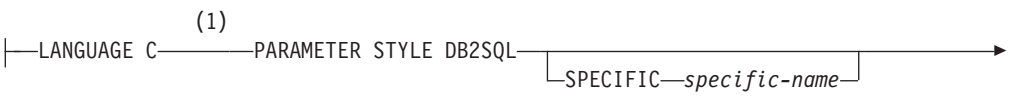
parameter-declaration:

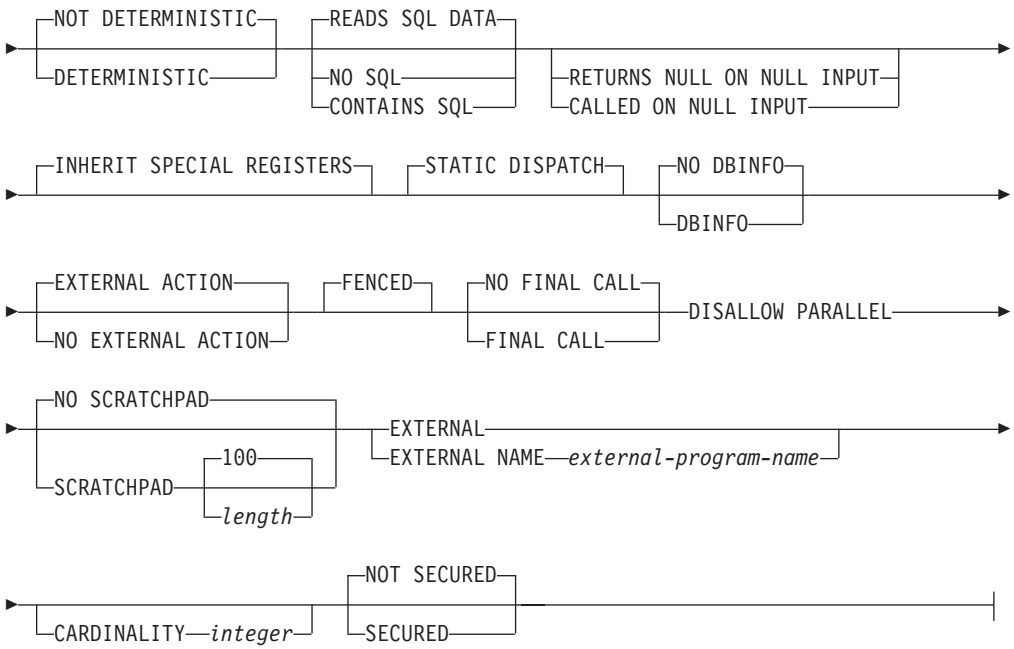
data-type1, data-type2:

built-in-type:



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

function-name
Names the user-defined function. If *function-name* is not qualified, it is implicitly qualified with the default schema name. The schema must be a valid schema name for functions, and the unqualified function name must not be any of the reserved function names. The same name can be used for more than one function in the same schema if the function signature of each function is unique. For more information about naming functions, see Choosing the schema and function name and Determining the uniqueness of functions in a schema.

(*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. 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 688. 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 "Attributes of the arguments of a routine program" on page 1136. 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 distinct type. The length, precision, or scale attributes a distinct type 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 734.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

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 data type or a distinct type that is based on a LOB data type. AS LOCATOR must not be specified if the function is defined with NO SQL.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

RETURNS TABLE
Specifies that the output of the function is a table. The parentheses that follow this clause enclose a list of names and the data types of the columns of the result table. For example: more than (247 - (*n**2))/2 columns must not be specified, where *n* is the number of columns of the result table.

column-name
Specifies the name of this column. The name must not be qualified and must not be the same as any other *column-name* in the RETURNS TABLE clause.

data-type2
Specifies the data type of the column. The column is nullable.

The same considerations that apply to the data type of input parameters, as described under data-type1, apply to the data type of the columns of the result table of the function.

AS LOCATOR
Specifies that the function returns a locator to the value rather than the actual value. Specify AS LOCATOR only if the column of the result table of the function has a LOB data type or a distinct type that is based on a LOB data type. AS LOCATOR must not be specified if the function is defined with NO SQL.

For more information on the AS LOCATOR clause, see Specifying AS LOCATOR for a parameter.

LANGUAGE
Specifies the language interface convention to which the body of the function is written. All programs must be designed to run in the server's environment.

C Specifies that the external function is written in C or C++. The database manager will invoke the function using the C language calling conventions.

PARAMETER STYLE
Specifies the conventions for passing parameters to and returning values from the function. For more information, see "Parameter passing for external routines" on page 1127.

DB2SQL
Specifies the parameter passing convention that supports passing null values both as input and for output. The parameters that are passed between the invoking SQL statement and the function include the following, in the order listed:

- *n* parameters for the input parameters that are specified for the function
- *m* parameters for the result columns of the function that are specified on the RETURNS TABLE clause
- *n* parameters for the 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
- The SQLSTATE that is to be returned to the database manager
- The qualified name of the function
- The specific name of the function
- The SQL diagnostic string that is to be returned to the database manager

The function passes from one to three additional arguments, depending on other options that are specified:

- The scratchpad, if SCRATCHPAD is specified
- The call type
- The DBINFO structure, if DBINFO is specified

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information on specific names, see Specifying a specific name for a function.

NOT DETERMINISTIC or 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 table each time that the function is invoked with the same input arguments even when the referenced data in the database has not changed. 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, provided that the referenced data in the database has not changed. The database manager uses this information during optimization of SQL statements.

The database manager does not verify that the function program is consistent with the specification of NOT DETERMINISTIC or DETERMINISTIC.

READS SQL DATA, NO SQL, or CONTAINS 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 "SQL statement data access classification for routines" on page 974. The default is READS SQL DATA.

READS SQL DATA
Specifies that function can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

NO SQL
Specifies that the function cannot execute any SQL statements.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access indication of CONTAINS SQL. The function cannot execute any SQL statements that read or modify data.

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 invoked if any of the input arguments is null. The result is an empty table (a table with no rows).

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 body of the function must be coded to test for null argument values. An empty table might be returned depending on the logic in the body of the function.

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 an additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application run-time authorization ID, the schema name, and identification of the database manager that invoked the function. For details about the argument and its structure, see "Database information in external routines (DBINFO)" on page 1138 or the sqludf include file.

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 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.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

The database manager does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

FENCED
Specifies that the external function runs in an environment that is isolated from the database manager environment.

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 that this call is the first call to the function for this reference to the function in this SQL statement. The scratch pad is initialized to zeroes.

Open call
Specifies that this call is a call to open the table function result in

this SQL statement. If NO FINAL CALL is specified, then the scratch pad is initialized to zeroes.

Fetch call
Specifies that this call is a call to fetch a row from the table function result in this SQL statement.

Close call
Specifies that this call is a call to close the table function result in this SQL statement.

Final call
Specifies that this call is the last call to the function, which enables 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 the step of closing the cursor.
- *Close of a cursor:* If a cursor that references the table function is defined as WITH HOLD, a final call is made when the cursor is closed.

If a commit, or rollback causes the final call, the function cannot issue any SQL statements except for the CLOSE statement.

DISALLOW PARALLEL
Specifies that the function cannot be run in parallel. Table functions cannot run in parallel.

NO SCRATCHPAD or SCRATCHPAD
Specifies whether the database manager provides a scratchpad for the function. A scratchpad provides an area for the function to save information from one invocation to the next. The default is NO SCRATCHPAD.

NO SCRATCHPAD
Specifies that a scratchpad is not allocated and passed to the function.

SCRATCHPAD *length*
Specifies that when the function is invoked for the first time, the database manager allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.
- The database manager initializes the scratchpad to all binary zeros (X'00').
- The scope of a scratchpad is the SQL statement. Each reference to the function in an SQL statement, has one scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, two scratch pads are allocated for the two references to UDFX in the following SQL statement:

```
SELECT *
  FROM TABLE (UDFX(A)),  TABLE (UDFX(B));
```

- The scratchpad is persistent. The database manager preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. The database manager initializes the scratchpads when it begins to

execute an SQL statement. The database manager does not reset scratchpads when a correlated subquery begins to execute.

- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify FINAL CALL to ensure that the database manager calls the function one more time so that the function can free those system resources.

Each time the function is invoked, the database manager passes an additional argument for the scratchpad.

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 the NAME clause is not specified, 'NAME *function-name*' is implicit. In this case, *function-name* must not be longer than 8 characters. For more information on the maximum length of an external program name, see Chapter 9, "SQL limits," on page 959.

NAME *external-program-name*
Specifies the program that is to be executed when the function is invoked. The executable form of the external program need not exist when the CREATE FUNCTION statement is executed. However, it must exist at the current server when the function is invoked.

CARDINALITY *integer*
Specifies an estimate of the expected number of rows to be returned by the function for the database manager to use during optimization. *integer* must be in the range from 0 to 2 147 483 647 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.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row access control and column access control.

NOT SECURED
Specifies that the function is considered not secure for row access control and column access control. This is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

A function must be defined as secure when it is referenced in a row permission or a column mask.

Notes

General considerations for defining user-defined functions: For general information on defining user-defined functions, see "CREATE FUNCTION" on page 606.

Owner privileges: The owner is authorized to execute the function (EXECUTE) and has the ability to grant these privileges to others. For more information on the EXECUTE privilege, see "GRANT (function or procedure privileges)" on page 793. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

Language considerations: A C program must be written to run as a subroutine.

For information on creating the programs for a function, see Chapter 19, "Coding programs for use by external routines," on page 1127.

Examples

Example 1: The following example 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 table function always returns the same table; therefore, it is defined as DETERMINISTIC. In addition, the DISALLOW PARALLEL clause is added because table functions cannot operate in parallel.

Although the size of the output for DOCMATCH is highly variable, CARDINALITY 20 is a representative value and is specified to help the database manager during optimization.

```
CREATE FUNCTION DOCMATCH (VARCHAR(30),VARCHAR(255))
    RETURNS TABLE ( DOC_ID CHAR(16))
    EXTERNAL NAME  ABC
    LANGUAGE C
    PARAMETER STYLE DB2SQL
    NO SQL
    DETERMINISTIC
    NO EXTERNAL ACTION
    FENCED
    SCRATCHPAD
    FINAL CALL
    DISALLOW PARALLEL
    CARDINALITY 20
```

CREATE FUNCTION (sourced)

The CREATE FUNCTION (sourced) statement defines a user-defined function that is based on an 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
- Database administrator authority.

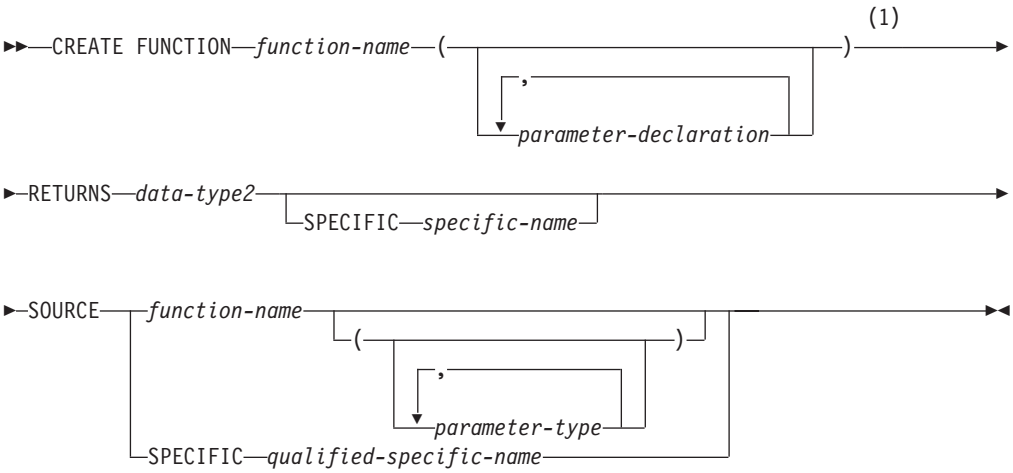
The privileges held by the authorization ID of the statement must include at least one of the following:

- The EXECUTE privilege for the function identified in the SOURCE clause
- Database administrator authority.

For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Database administrator authority.

Syntax

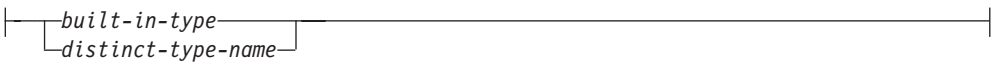


parameter-declaration:

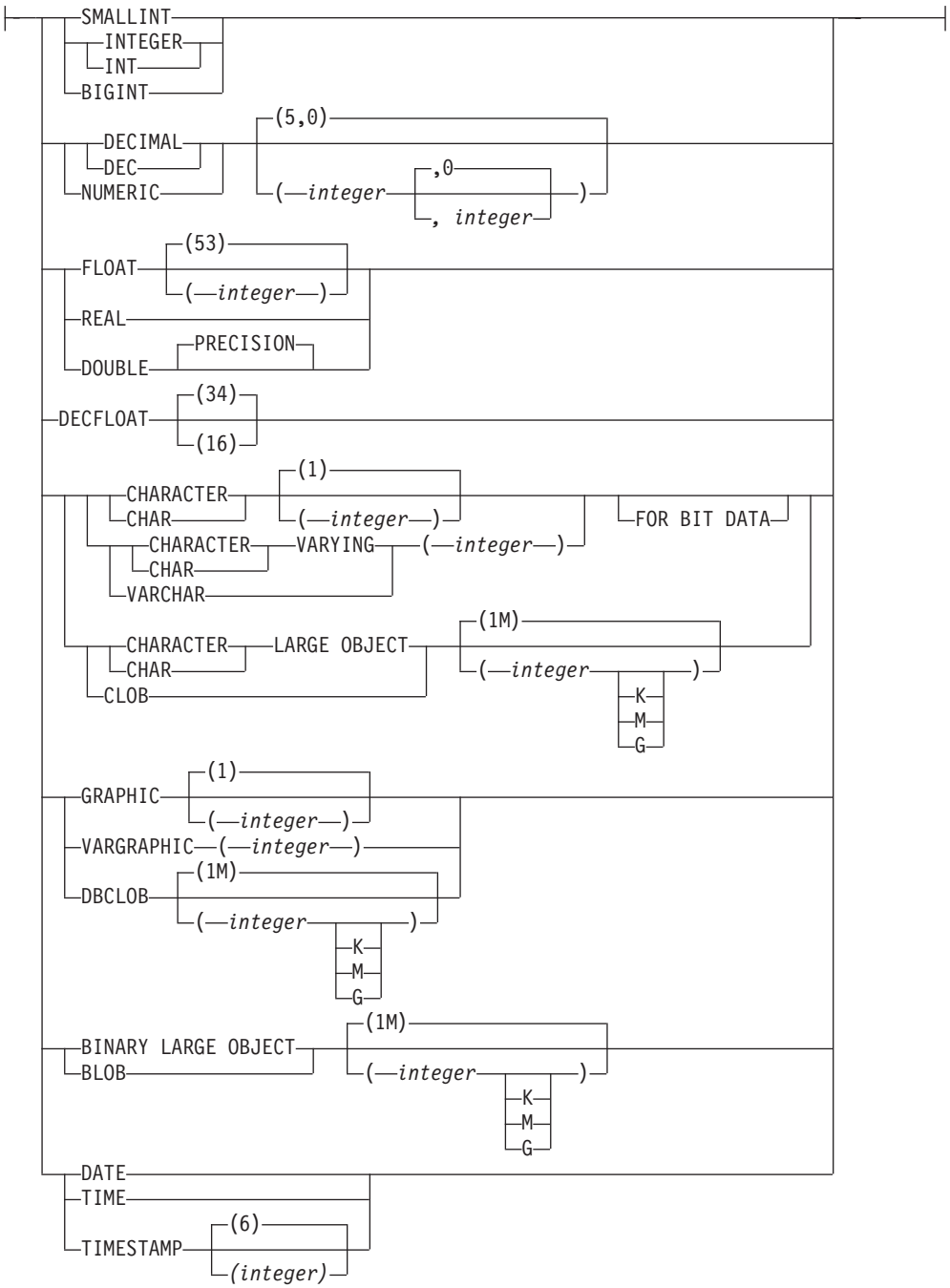
Notes:

1 RETURNS, SPECIFIC, and SOURCE can be specified in any order.

data-type1, data-type2, data-type3:



built-in-type:



parameter-type:

Notes:

1 Empty parentheses are allowed for some data types specified in this context.

Description

function-name
Names the user-defined function. If *function-name* is not qualified, it is implicitly qualified with the default schema name. The schema must be a valid schema name for functions, and the unqualified function name must not be any of the reserved function names. The same name can be used for more than one function in the same schema if the function signature of each function is unique. For more information about naming functions, see Choosing the schema and function name and Determining the uniqueness of functions in a schema.

(*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. 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 type.

Any valid SQL data type can be used, provided that it is castable to the type of the corresponding parameter of the function that is identified in the SOURCE clause. (For more information, see “Casting between data types” on page 74.) 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
Specifies a built-in data type. For a more complete description of each built-in data type, see “CREATE TABLE” on page 688.

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 734.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

RETURNS
Specifies the result of the function.

data-type2
Specifies the data type of the result. The output parameter is nullable.

The data type can be a built-in data type or distinct type. The data type of the final result of the source function must match or be castable to the result of the sourced function. (For information about casting data types, see "Casting between data types" on page 74.) 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.

SPECIFIC *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 is 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
- BLOB when more than one argument is specified
- CARDINALITY
- CHAR when more than one argument is specified
- CHARACTER_LENGTH
- CLOB when more than one argument is specified
- COALESCE
- CONTAINS
- COUNT(*)
- COUNT_BIG(*)
- DBCLOB when more than one argument is specified
- DECFLOAT when more than one argument is specified
- DECIMAL when more than one argument is specified
- DECRYPT_BIT and DECRYPT_CHAR
- EXTRACT
- GRAPHIC when more than one argument is specified
- LPAD when more than two arguments are specified
- MAX
- MAX_CARDINALITY

- MIN
- NULLIF
- POSITION
- RAISE_ERROR
- RID
- RPAD when more than two arguments are specified
- SCORE
- SECOND when more than one argument is specified
- STRIP
- SUBSTRING
- TIMESTAMP when the second argument is an integer
- TIMESTAMP_FORMAT when more than two arguments are specified
- TRANSLATE when more than one argument is specified
- TRIM
- TRIM_ARRAY
- VALUE
- VARCHAR when more than one argument is specified
- VARGRAPHIC when more than one argument is specified
- VERIFY_GROUP_FOR_USER when more than two arguments are specified
- XMLAGG
- XMLATTRIBUTES
- XMLCOMMENT
- XMLCONCAT
- XMLDOCUMENT
- XMLELEMENT
- XMLFOREST
- XMLNAMESPACES
- XMLPARSE
- XMLPI
- XMLSERIALIZE
- XMLTEXT
- XSLTRANSFORM

function-name
Identifies the function that is to be used as the source function. 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 that is 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. The specified parameters must match the data types in the

corresponding position that were specified when the function was created. The database manager uses the number of data types and the logical concatenation of the data types to identify the specific function instance. Synonyms for data types are considered a match.

If *function-name*() is specified, the identified function 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, DECIMAL() is considered a match for a parameter of a function that is defined with a data type of DECIMAL(7,2). However, FLOAT cannot be specified with empty parentheses 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 need 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.

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 distinct type that is based on a LOB.

SPECIFIC *qualified-specific-name*
Identifies the particular user-defined function that is used as the source function by its specific name. The *qualified-specific-name* must identify a specific user-defined function that exists in the specified or implicit 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.

Notes

General considerations for defining user-defined functions: For general information on defining user-defined functions, see "CREATE FUNCTION" on page 606.

Owner privileges: The owner is authorized to execute the function (EXECUTE).

- If the underlying function is a user-defined function, and the owner is authorized with the grant option to execute the underlying function, then the privilege on the new function includes the grant option. Otherwise, the owner can execute the new function but cannot grant others the privilege to do so.
- If the underlying function is a built-in function, the owner is authorized with the grant option to execute the underlying built-in function and the privilege on the new function includes the grant option.

For more information on the EXECUTE privilege for functions, see "GRANT (function or procedure privileges)" on page 793. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

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 defined by the options specified implicitly or explicitly when the external scalar function was created. 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 that are specified on

the EXTERNAL clause of the CREATE FUNCTION statement for function C.

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, 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)

The CREATE FUNCTION (SQL scalar) statement defines an SQL scalar function at the current server. A user-defined SQL scalar function returns a single value each time it is invoked. The body of the function is written in the SQL procedural language.

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
- Database administrator authority.

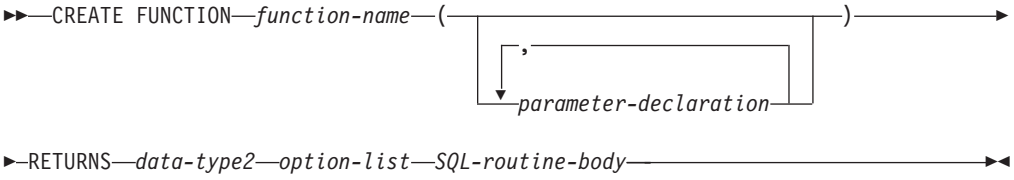
If the SECURED option is specified:

- The authorization ID of the statement must have Security administrator authority.

For each distinct type referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Database administrator authority.

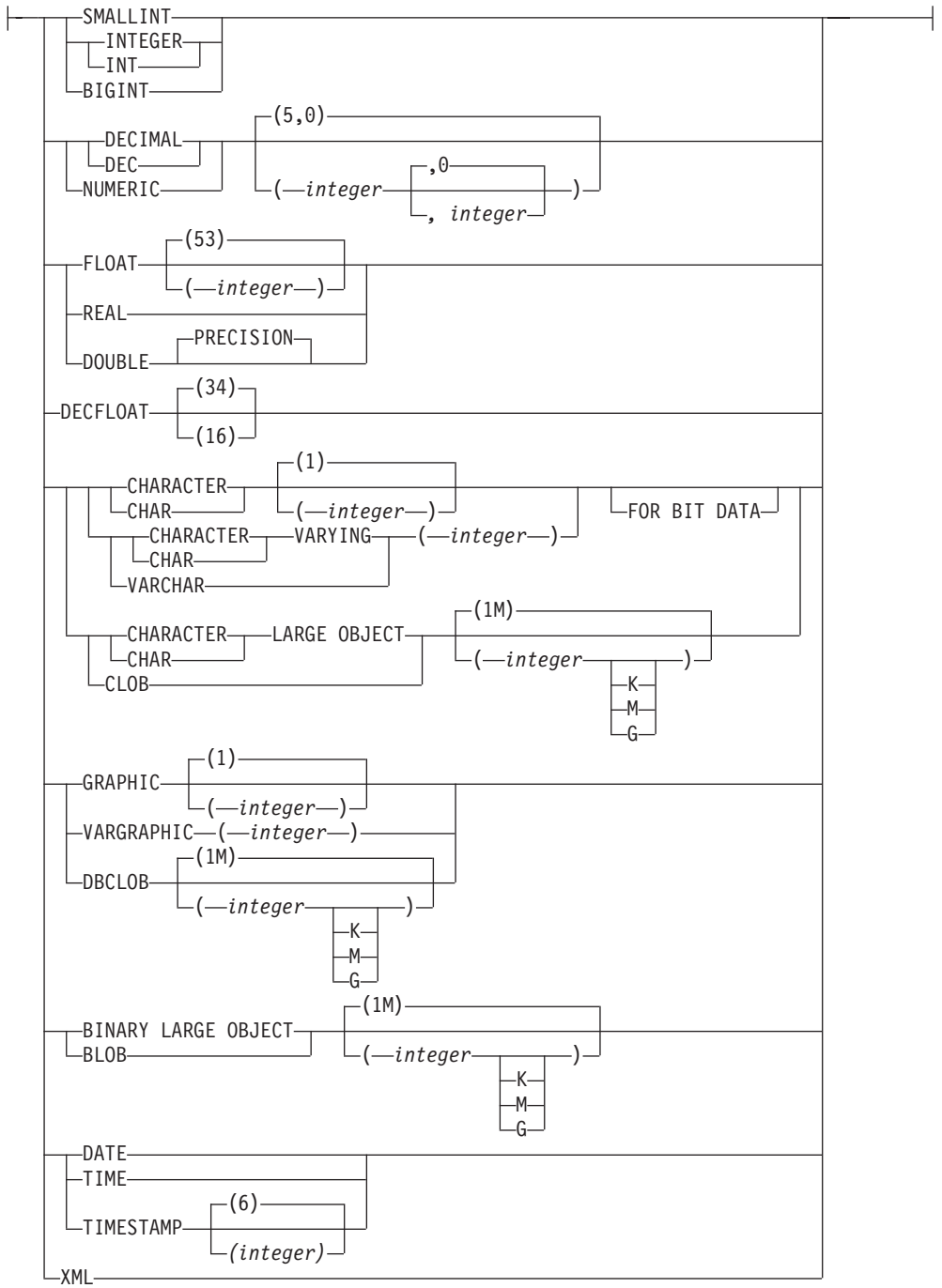
Syntax



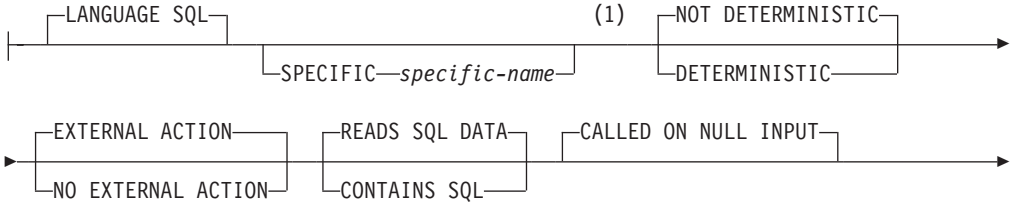
parameter-declaration:

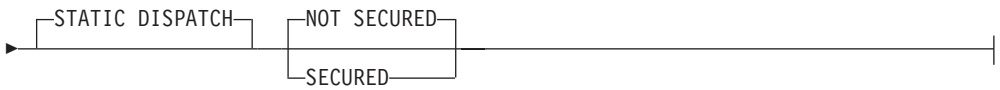
data-type1, data-type2:

built-in-type:

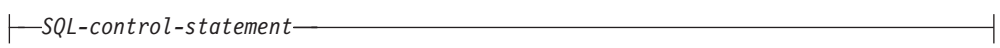

SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
(5,0)
(integer
,0
, integer
)
FLOAT
(53)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
FOR BIT DATA
CHARACTER
CHAR
VARYING
(integer)
VARCHAR
CHARACTER
CHAR
LARGE OBJECT
CLOB
(1M)
(integer
K
M
G
)
GRAPHIC
(1)
(integer)
VARGRAPHIC
(integer)
DBCLOB
(1M)
(integer
K
M
G
)
BINARY LARGE OBJECT
BLOB
(1M)
(integer
K
M
G
)
DATE
TIME
TIMESTAMP
(6)
(integer)
XML

option-list:


LANGUAGE SQL
SPECIFIC
specific-name
(1)
NOT DETERMINISTIC
DETERMINISTIC
EXTERNAL ACTION
NO EXTERNAL ACTION
READS SQL DATA
CONTAINS SQL
CALLED ON NULL INPUT



SQL-routine-body:



Notes:

1 This clause and the clauses that follow in the *option-list* can be specified in any order. Each clause may be specified at most once.

Description

function-name
Names the user-defined function. If *function-name* is not qualified, it is implicitly qualified with the default schema name. The schema must be a valid schema name for functions, and the unqualified function name must not be any of the reserved function names. The same name can be used for more than one function in the same schema if the function signature of each function is unique. For more information about naming functions, see Choosing the schema and function name and Determining the uniqueness of functions in a schema.

(*parameter-declaration***,...)**
Specifies the number of input parameters of the function and the name and 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. For more information, see Defining the parameters.

parameter-name
Names the input 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 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 688.

distinct-type-name
Specifies a distinct type. The length, precision, or scale attributes for a distinct type 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 734.

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.

RETURNS
Specifies the result of the function.

data-type2
Specifies the data type of the result. The output parameter is nullable.

Similar considerations that apply to the data type of the input parameters, as described in Defining the parameters, apply to the data type of the result of the function.

LANGUAGE SQL
Specifies that the function is written exclusively in SQL.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information, see Specifying a specific name for a function.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results for identical 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.

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. An example of a function that is deterministic is one that calculates the square root of the input argument.

The body of the SQL routine must be consistent with the implicit or explicit specification of DETERMINISTIC or NOT DETERMINISTIC. A function that is defined as DETERMINISTIC must not invoke another function that is defined as NOT DETERMINISTIC, nor can it reference a special register. For example, an SQL function that invokes the RAND built-in function in its RETURN statement must have been created as a non-deterministic function.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function can take 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 file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function can cause a change to the state of an object that the database manager does not manage.

A function with external actions might return incorrect results if the function is executed by parallel tasks. For example, if a function sends a

note for each initial call to that function, one note is sent for each parallel task instead of one note for the function.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

EXTERNAL ACTION must be implicitly or explicitly specified if the SQL routine body invokes a function that is defined with EXTERNAL ACTION.

READS SQL DATA or CONTAINS SQL
Specifies the classification of SQL statements that the function can execute. The database manager verifies that the SQL statements that the function issues is consistent with this specification. For the classification of each statement, see "SQL statement data access classification for routines" on page 974. The default is READS SQL DATA.

READS SQL DATA
Specifies that function can execute statements with a data access classification of READS SQL DATA or CONTAINS 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. The function cannot execute any SQL statements that read or modify data. For example, READS SQL DATA (or MODIFIES SQL DATA) must be specified if the body of the SQL function contains a subselect or if it invokes a function that can read data.

CALLED ON NULL INPUT
Specifies that the function is invoked regardless of whether any of the input arguments is null. This specification means that the function is responsible for testing for null argument values. The function can return a null or nonnull value.

STATIC DISPATCH
Specifies that the function is dispatched statically. All functions are statically dispatched.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row access control and column access control.

NOT SECURED
Specifies that the function is considered not secure for row access control and column access control. This is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

A function must be defined as secure when it is referenced in a row permission or a column mask.

SQL-routine-body
Specifies the body of the function.

SQL-control-statement
Specifies a single SQL control statement, including a compound statement. See Chapter 8, "SQL control statements," on page 907 for more information about defining SQL functions.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, COMMIT, and ROLLBACK 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 last when the function is invoked.

Notes

General considerations for defining user-defined functions: For general information on defining user-defined functions, see "CREATE FUNCTION" on page 606.

Owner privileges: The owner is authorized to execute the function (EXECUTE). The EXECUTE privilege can be granted to others only if the owner has the authority to grant the EXECUTE privilege on every user-defined function and the USAGE privilege on every sequence reference referenced in the RETURN statement of the body of the function. For more information, see "GRANT (function or procedure privileges)" on page 793. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

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.

Examples

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
- Database administrator authority.

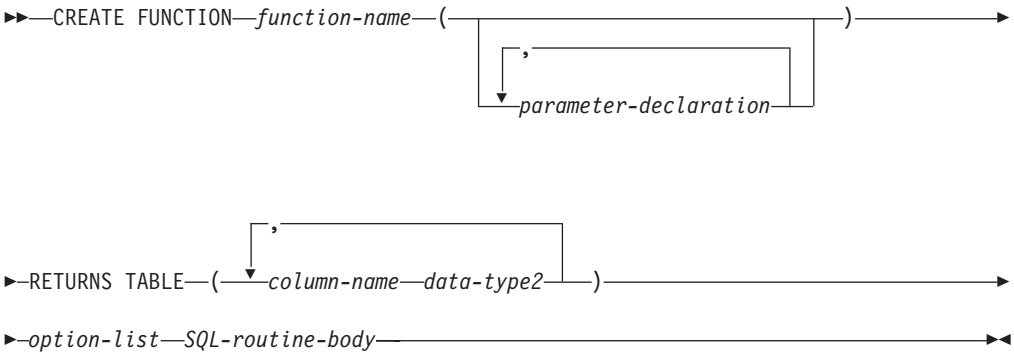
If the SECURED option is specified:

- The authorization ID of the statement must have Security administrator authority.

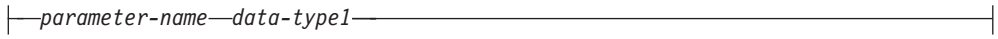
For each distinct type referenced in the statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Database administrator authority.

Syntax

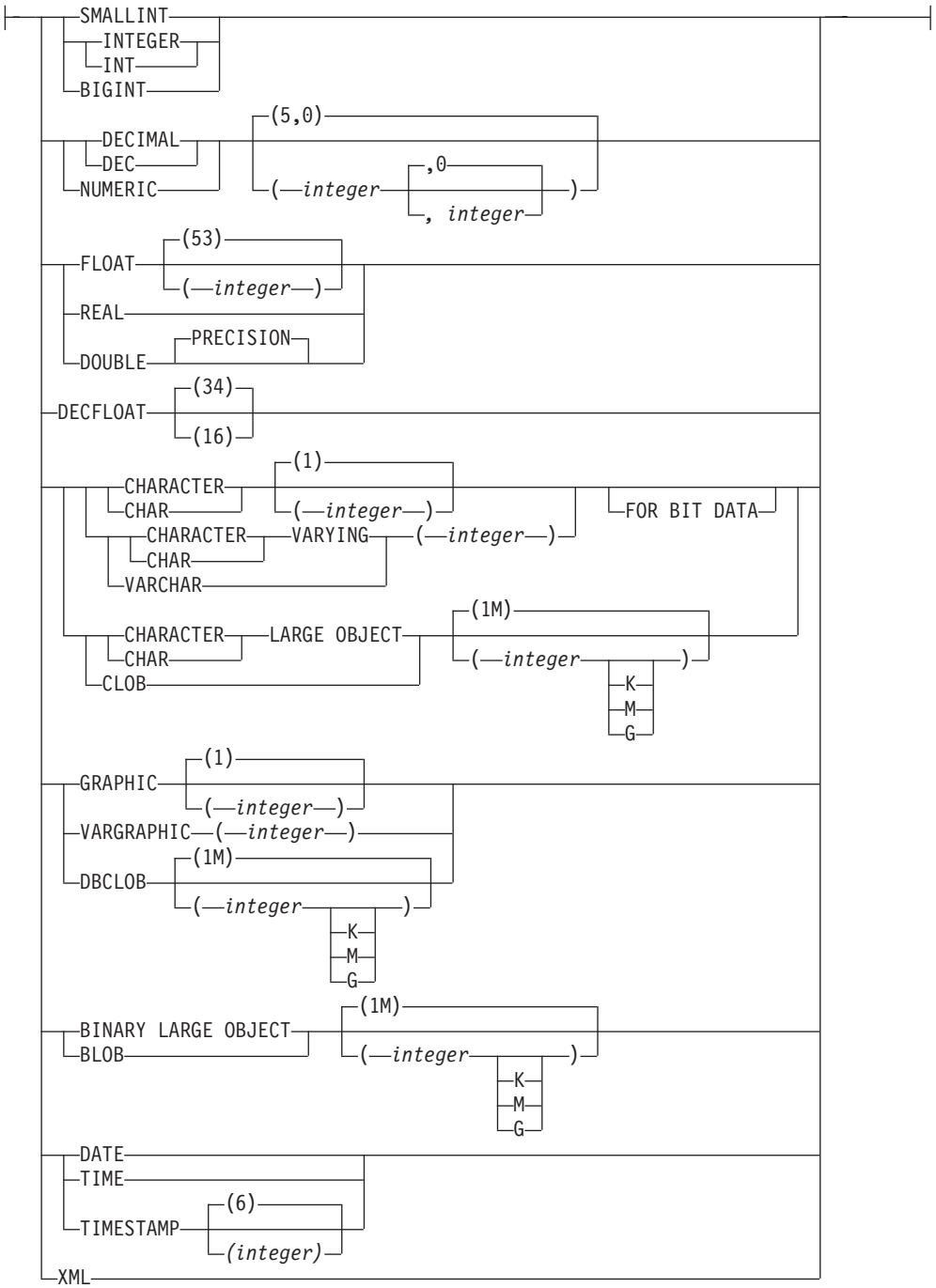


parameter-declaration:

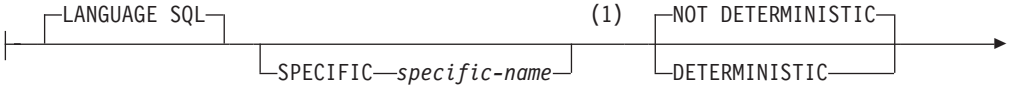


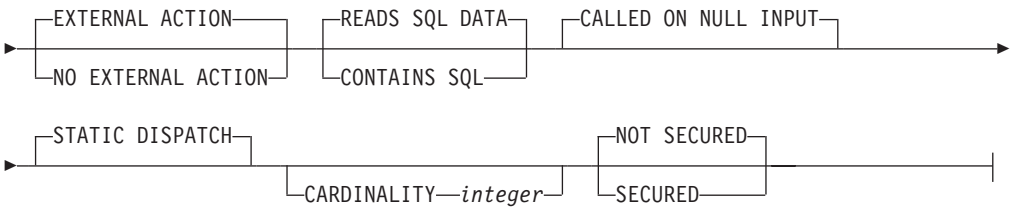
data-type1, data-type2:

built-in-type:



option-list:





SQL-routine-body:

Notes:

1 This clause and the clauses that follow in the *option-list* can be specified in any order. Each clause may be specified at most once.

Description

function-name
Names the user-defined function. If *function-name* is not qualified, it is implicitly qualified with the default schema name. The schema must be a valid schema name for functions, and the unqualified function name must not be any of the reserved function names. The same name can be used for more than one function in the same schema if the function signature of each function is unique. For more information about naming functions, see Choosing the schema and function name and Determining the uniqueness of functions in a schema.

(*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. 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-declaration* 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 688.

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 734.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

RETURNS TABLE
Specifies the output table of the function.

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 or a distinct type. When the function is invoked the results are assigned to these data types (using storage assignment rules).

LANGUAGE SQL
Specifies that the function is written exclusively in SQL.

SPECIFIC *specific-name*
Specifies a unique name for the function. For more information, see Specifying a specific name for a function.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results for identical 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.

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. An example of a function that is deterministic is one that calculates the square root of the input argument.

The body of the SQL routine must be consistent with the implicit or explicit specification of DETERMINISTIC or NOT DETERMINISTIC. A function that is defined as DETERMINISTIC must not invoke another function that is defined as NOT DETERMINISTIC, nor can it reference a special register. For example, an SQL function that invokes the RAND built-in function in its RETURN statement must have been created as a non-deterministic function.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function can take 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 file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function can cause a change to the state of an object that the database manager does not manage.

A function with external actions might return incorrect results if the function is executed by parallel tasks. For example, if a function sends a note for each initial call to that function, one note is sent for each parallel task instead of one note for the function.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

EXTERNAL ACTION must be implicitly or explicitly specified if the SQL routine body invokes a function that is defined with EXTERNAL ACTION.

READS SQL DATA or CONTAINS SQL
Specifies the classification of SQL statements that the function can execute. The database manager verifies that the SQL statements that the function issues is consistent with this specification. For the classification of each statement, see "SQL statement data access classification for routines" on page 974. The default is READS SQL DATA.

READS SQL DATA
Specifies that function can execute statements with a data access classification of READS SQL DATA or CONTAINS 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. The function cannot execute any SQL statements that read or modify data. For example, READS SQL DATA (or MODIFIES SQL DATA) must be specified if the body of the SQL function contains a subselect or if it invokes a function that can read data.

CALLED ON NULL INPUT
Specifies that the function is invoked regardless of whether any of the input arguments is null. This specification means that the function is responsible for testing for null argument values. The function can return a null or nonnull value.

STATIC DISPATCH
Specifies that the function is dispatched statically. All functions are statically dispatched.

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.

SQL-routine-body
Specifies the body of the function.

RETURN-statement
Specifies the return value of the function. For description of the statement, see "RETURN statement" on page 951. Parameter names can be referenced in the RETURN statement.

ATOMIC
Specifies that an unhandled exception condition within the RETURN statement causes the statement to be rolled back.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row access control and column access control.

NOT SECURED
Specifies that the function is considered not secure for row access control and column access control. This is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

A function must be defined as secure when it is referenced in a row permission or a column mask.

Notes

General considerations for defining user-defined functions: For general information on defining user-defined functions, see "CREATE FUNCTION" on page 606.

Owner privileges: The owner is authorized to execute the function (EXECUTE). The EXECUTE privilege can be granted to others only if the owner has the authority to grant the EXECUTE privilege on every user-defined function and the USAGE privilege on every sequence reference referenced in the RETURN statement of the body of the function. For more information, see "GRANT (function or procedure privileges)" on page 793. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

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.

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 defines 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

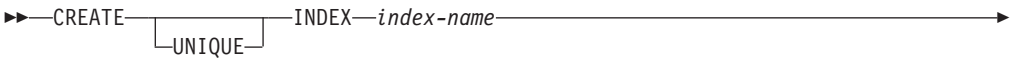
The privileges held by the authorization ID of the statement must include at least one of the following:

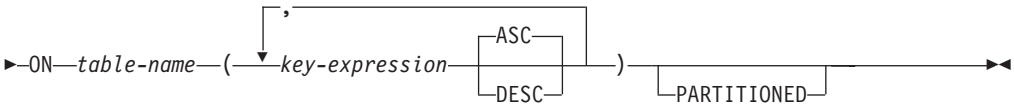
- The privilege to create in the schema
- Database administrator authority.

The privileges held by the authorization ID of the statement must include at least one of the following:

- The INDEX privilege for the table
- Database administrator authority.

Syntax





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 *key-expression* 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 and an error is returned.

index-name

Names the index. The name, including the implicit or explicit qualifier, must not identify an index, table, view, or alias that already exists at the current server. The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT.

The implicit or explicit qualifier for indexes on declared temporary tables must be SESSION.

ON *table-name*
Identifies the table on which the index is to be created. The name must identify a base table that exists at the current server, but it must not identify a catalog table.

(*key-expression***,...)**
Identifies the list of expressions or columns that will be part of the index key.

Each column referenced in the *key-expression* must be an unqualified name that identifies a column of the table. The *key-expression* must not be specified more than once. The data type of any referenced column and the result data type of the *key-expression* cannot be:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

The expression must contain at least one column reference.

It must not contain any of the following:

- Subqueries
- Aggregate functions
- Variables
- Global variables
- Parameter markers
- Special registers and built-in functions that depend on the value of a special register
- Sequence references
- A CASE expression
- OLAP specifications
- 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
- If *key-expression* references the TRANSLATE function, the function invocation must contain the *to-string* argument.
- The following built-in scalar functions:

ATAN2	GETHINT	RAND	TRUNC_TIMESTAMP
CARDINALITY	IDENTITY_VAL_LOCAL	REPEAT	VARCHAR_FORMAT
CONTAINS	INSERT	REPLACE	VERIFY_GROUP_FOR_USER
DAYNAME	LPAD	ROUND_TIMESTAMP	WEEK_ISO
DECRYPT_BIT	MAX_CARDINALITY	RPAD	XMLPARSE
DECRYPT_CHAR	MONTHNAME	SCORE	XSLTRANSFORM
DIFFERENCE	MONTHS_BETWEEN	SOUNDEX	
ENCRYPT	NEXT_DAY	TIMESTAMP_FORMAT	
GENERATE_UNIQUE	RAISE_ERROR	TIMESTAMPDIFF	

The number of specified *key-expression*s must not exceed 64 and the sum of their length attributes must not exceed 2000. See Table 63 on page 967 for more information. Note that this length can be reduced by system overhead which varies according to the data type of the *key-expression* and whether it is nullable. See Byte Counts for more information on overhead affecting this limit.

ASC
Specifies that index entries are to be kept in ascending order of the expression values. This is the default.

DESC
Specifies that index entries are to be kept in descending order of the expression values.

Ordering is performed in accordance with the comparison rules described in "Assignments and comparisons" on page 77. The null value is higher than all other values.

PARTITIONED
Specifies that an index partition should be created for each data partition defined for the table using the specified expressions. The *table-name* must identify a partitioned table. If the index is unique, the *key-expression*s of the index must be the same or a superset of the columns of the data partition key. If the index is not unique and the table is partitioned, whether the index is partitioned is product-specific.

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.

Owner privileges: There are no specific privileges on an index. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

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)
```

CREATE MASK

The CREATE MASK statement creates a column mask for column access control at the current server. A column mask specifies what value should be returned for the specified column.

Invocation

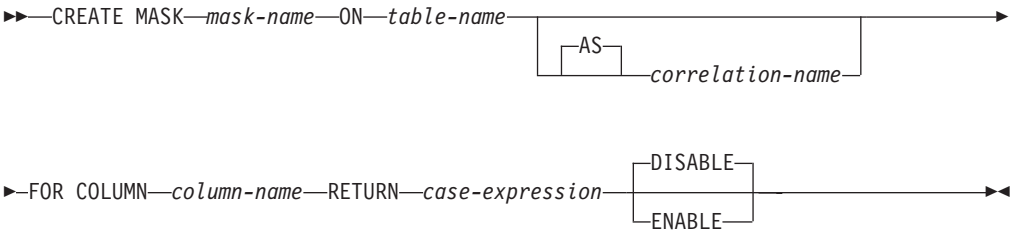
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The authorization ID of the statement must have security administrator authority.

Additional privileges are not needed to reference other objects in the mask definition. For example, the SELECT privilege is not needed to retrieve from a table, and the EXECUTE privilege is not needed to call a user-defined function.

Syntax



Description

mask-name
Names the column mask for column access control. The name, including the implicit or explicit qualifier, must not be the same as a column mask or a row permission that already exists at the current server. The *mask-name* cannot start with QIBM.

The schema name for the *mask-name* must be the same as the schema name for *table-name*.

table-name
Identifies the table on which the column mask is created. The name must identify a table that exists at the current server. It must not identify a declared temporary table, view, alias, materialized query table or table that is directly or indirectly referenced in the definition of a materialized query table, or catalog table.

correlation-name
Specifies a correlation name that can be used within *case-expression* to designate the table.

FOR COLUMN *column-name*
Identifies the column to which the mask applies. It must be an unqualified name that identifies a column of the table. A mask must not already exist for the column. The column must not be any of the following columns:

- A LOB column or a distinct type column that is based on a LOB.

- An XML column.
- A column referenced in an expression that defines a generated column.

RETURN *case-expression*
Specifies a CASE expression to be evaluated to determine the value to return for the column. The result of the CASE expression is returned in place of the column value in a row. The result data type, length, null attribute, and CCSID of the CASE expression must be compatible with the data type of the column. For more information about the compatibility of data types see "Assignments and comparisons" on page 77. If the data type of *column-name* is a user defined data type, the result data type of the CASE expression must be the same user defined type. Any objects referenced in the case expression must exist at the current server. The case expression must not reference any of the following:

- The table for which the column mask is being defined
- A declared global temporary table
- A variable (host variable, SQL variable, SQL parameter, or trigger transition variable)
- A parameter marker
- A user-defined function that is defined as NOT SECURED
- A function that is not deterministic, has an external action, or modifies SQL data
- An OLAP specification
- A ROW CHANGE expression
- A sequence reference
- A * or *name*.* in a select clause
- A table function
- An aggregate function, unless it is specified in a subquery
- A collection-derived table (UNNEST)
- A LOB column or a distinct type column that is based on a LOB
- An XML column
- A view that contains any of the above

If the CASE expression references tables for which row or column access control is active, access controls for those tables are not cascaded.

ENABLE or DISABLE
Specifies that the column mask is to be initially enabled or disabled for column access control.

DISABLE
Specifies that the column mask is to be disabled for column access control. The column mask will remain ineffective regardless of whether column access control is activated for the table or not. This is the default.

ENABLE
Specifies that the column mask is to be enabled for column access control. If column access control is not currently activated for this table, the column mask will become effective when column access control is activated for the table. If column access control is currently activated for the table, the column mask becomes effective immediately.

Notes

How column masks affect queries: The application of enabled column masks does not interfere with the operations of other clauses within the statement such as WHERE, GROUP BY, HAVING, SELECT DISTINCT, or ORDER BY. The rows that are returned in the final result table remain the same, except that the values in the resulting rows might have been masked by the column masks. As such, if the masked column also appears in an ORDER BY clause with a sort-key expression, the order is based on the original values of the column and the masked values in the final result table might not reflect that order. Similarly, the masked values might not reflect the uniqueness enforced by a SELECT DISTINCT statement. If the masked column is embedded in an expression, the result of the expression might become different because the column mask is applied on the column before the expression evaluation can take place. For example, a column mask on column SSN might change the result of the aggregate function COUNT(DISTINCT SSN) because the DISTINCT operation is performed on the masked values. However, if the expression in the query is the same as the expression that is used to mask the column value in the definition of the column mask, the result of the expression might remain unchanged. For example, the expression in the query is 'XXX-XX-' || SUBSTR(SSN, 8, 4) and the same expression appears in the definition of the column mask. In this particular example, you can replace the expression in the query with column SSN to avoid the same expression being evaluated twice.

Conflicts between the definition of a column mask and SQL: A column mask is created as a stand alone object, without knowing all of the contexts in which it might be used. To mask the value of a column in the final result table, the definition of the column mask is merged into a query by DB2. When the definition of the column mask is brought into the context of the statement, it might conflict with certain SQL semantics in the statement. Therefore, in some situations, the combination of the statement and the application of the column mask can return an error. When this happens, either the statement needs to be modified or the column mask must be dropped or recreated with a different definition.

Column masks and null columns: If the column is not nullable, the definition of its column mask will not, most likely, consider a null value for the column. After the column access control is activated for the table, if the table is the null-padded table in an outer join, the value of the column in the final result table might be a null. To ensure that the column mask can mask a null value, if the table is the null-padded table in an outer join, DB2 will add "WHEN target-column IS NULL THEN NULL" as the first WHEN clause to the column mask definition. This forces a null value to always be masked as a null value. For a nullable column, this removes the ability to mask a null value as something else. Example 5 shows this added WHEN clause.

Column mask values for SQL data change statements: When columns are used to derive new values for an INSERT, UPDATE, MERGE, or a SET transition-variable assignment statement, the original values of the column, not the masked values, are used to derive the new values. If the columns have column masks, those column masks are applied to ensure that the evaluation of the access control rules at run time masks the column to itself, not to a constant or an expression. This is to ensure that the masked values are the same as the original column values. If a column mask does not mask the column to itself, the existing row is not updated or the new row is not inserted and an error is returned at run time. The rules that are used to apply column masks in order to derive the new values follow the same rules for the final result table of a query. See the data change statements for how the column masks are used to affect the insertability and updatability.

Column masks that are created before column access control is activated: The CREATE MASK statement is an independent statement that can be used to create a column access control mask before column access control is activated for a table. The only requirement is that the table and the columns exist before the mask is created. Multiple column masks can be created for a table but a column can have one mask only. The definition of a mask is stored in the DB2 catalog. Dependency on the table for which the mask is being created and dependencies on other objects referenced in the definition are recorded. A column mask can be created as enabled or disabled for column access control. An enabled column mask does not take effect until the ALTER TABLE statement with the ACTIVATE COLUMN ACCESS CONTROL clause is used to activate column access control for the table. A disabled column mask remains ineffective even when column access control is activated for the table. The ALTER MASK statement can be used to alter between ENABLE and DISABLE. After column access control is activated for a table, when the table is referenced in a data manipulation statement, all enabled column masks that have been created for the table are implicitly applied by DB2 to mask the values returned for the columns referenced in the final result table of the queries or to determine the new values used in the data change statements. Creating column masks before activating column access control for a table is the recommended sequence.

Column masks that are created after column access control is activated: The enabled column masks become effective as soon as they are committed. Thereafter, when the table is referenced in a data manipulation statement, all enabled column masks are implicitly applied by DB2 to the statement. Any disabled column mask remains ineffective even when column access control is activated for the table.

No cascaded effect when column or row access control enforced tables are referenced in column mask definitions: A column mask definition may reference tables and columns that are currently enforced by row or column access control. Access control from those tables and columns is ignored when the table for which the column mask is being created is referenced in a data manipulation statement.

Column masks that return data which is not assignable to the column the mask is defined on: A column mask can be defined so it can return data which is not assignable to the data type of the column the mask is defined on. When this occurs, the CREATE MASK statement is successful but a cast error will be reported when the mask is applied in a user query.

Examples

Example 1: After column access control is activated for table EMPLOYEE, Paul from the payroll department can see the social security number of the employee whose employee number is 123456. Mary who is a manager can see only the last four characters of the social security number. Peter who is neither cannot see the social security number.

```
CREATE MASK SSN_MASK ON EMPLOYEE
   FOR COLUMN SSN RETURN
      CASE
         WHEN (VERIFY_GROUP_FOR_USER(SESSION_USER,'PAYROLL') = 1)
            THEN SSN
         WHEN (VERIFY_GROUP_FOR_USER(SESSION_USER,'MGR') = 1)
            THEN 'XXX-XX-' || SUBSTR(SSN,8,4)
         ELSE NULL
      END
   ENABLE;

COMMIT;
```

```
ALTER TABLE EMPLOYEE
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT SSN FROM EMPLOYEE
   WHERE EMPNO = 123456;
```

Example 2: In the SELECT statement, column SSN is embedded in an expression that is the same as the expression used in the column mask SSN_MASK. After column access control is activated for table EMPLOYEE, the column mask SSN_MASK is applied to column SSN in the SELECT statement. For this particular expression, the SELECT statement produces the same result as before column access control is activated for all users. The user can replace the expression in the SELECT statement with column SSN to avoid the same expression getting evaluated twice.

```
CREATE MASK SSN_MASK ON EMPLOYEE
   FOR COLUMN SSN RETURN
      CASE
         WHEN (1 = 1)
             THEN 'XXX-XX-' || SUBSTR(SSN,8,4)
         ELSE NULL
      END
   ENABLE;

COMMIT;

ALTER TABLE EMPLOYEE
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT 'XXX-XX-' || SUBSTR(SSN,8,4) FROM EMPLOYEE
   WHERE EMPNO = 123456;
```

Example 3: Employee with EMPNO 123456 earns bonus $8000 and salary $80000 in May. When the manager retrieves his salary, the manager receives his salary, not the null value. This is because of no cascaded effect when column mask SALARY_MASK references column BONUS for which column mask BONUS_MASK is defined.

```
CREATE MASK SALARY_MASK ON EMPLOYEE
   FOR COLUMN SALARY RETURN
      CASE
         WHEN (BONUS < 10000)
            THEN SALARY
         ELSE NULL
      END
   ENABLE;

COMMIT;

CREATE MASK BONUS_MASK ON EMPLOYEE
   FOR COLUMN BONUS RETURN
      CASE
         WHEN (BONUS > 5000)
            THEN NULL
         ELSE BONUS
      END
   ENABLE;

COMMIT;
```

```
ALTER TABLE EMPLOYEE
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT SALARY FROM EMPLOYEE
   WHERE EMPNO = 123456;
```

Example 4: This example shows that DB2 adds "WHEN target-column IS NULL THEN NULL" as the first WHEN clause to the column mask definition then merges the column mask definition into the statement.

```
CREATE TABLE EMPLOYEE (EMPID INT,
                       DEPTID CHAR(8),
                       SALARY DEC(9,2) NOT NULL,
                       BONUS DEC(9,2));

CREATE MASK SALARY_MASK ON EMPLOYEE
   FOR COLUMN SALARY RETURN
      CASE
         WHEN SALARY < 10000
            THEN CAST(SALARY*2 AS DEC(9,2))
         ELSE COALESCE(CAST(SALARY/2 AS DEC(9,2)), BONUS)
      END
   ENABLE;

COMMIT;

CREATE MASK BONUS_MASK ON EMPLOYEE
   FOR COLUMN BONUS RETURN
      CASE
         WHEN BONUS > 1000
            THEN BONUS
         ELSE NULL
      END
   ENABLE;

COMMIT;

ALTER TABLE EMPLOYEE
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT SALARY FROM DEPT
   LEFT JOIN EMPLOYEE ON DEPTNO = DEPTID;

/* When SALARY_MASK is merged into the above statement,
* 'WHEN SALARY IS NULL THEN NULL' is added as the
* first WHEN clause, as follows:
*/

SELECT CASE WHEN SALARY IS NULL THEN NULL
            WHEN SALARY < 10000 THEN CAST(SALARY*2 AS DEC(9,2))
            ELSE COALESCE(CAST(SALARY/2 AS DEC(9,2)), BONUS)
       END SALARY
  FROM DEPT
    LEFT JOIN EMPLOYEE ON DEPTNO = DEPTID;
```

CREATE PERMISSION

The CREATE PERMISSION statement creates a row permission for row access control at the current server. It determines the rows within a table that are available based on the result of the *search-condition*.

Invocation

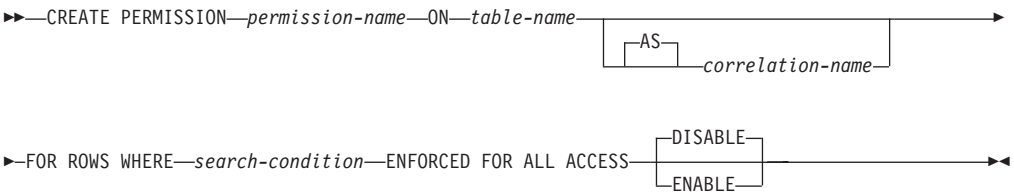
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The authorization ID of the statement must have security administrator authority.

Additional privileges are not needed to reference other objects in the permission definition. For example, the SELECT privilege is not needed to retrieve from a table, and the EXECUTE privilege is not needed to call a user-defined function.

Syntax



Description

permission-name
Names the row permission for row access control. The name, including the implicit or explicit qualifier, must not be the same as a column mask or a row permission that already exists at the current server. The *permission-name* cannot start with QIBM.

The schema name for the *permission-name* must be the same as the schema name for *table-name*.

table-name
Identifies the table on which the column permission is created. The name must identify a table that exists at the current server. It must not identify a declared temporary table, view, alias, materialized query table or table that is directly or indirectly referenced in the definition of a materialized query table, or catalog table.

correlation-name
Specifies a correlation name that can be used within *search-condition* to designate the table.

FOR ROWS WHERE
Indicates that a row permission is created. A row permission specifies a search condition under which rows of the table can be accessed.

search-condition
Specifies a condition that can be true, false, or unknown for a row of the table.

The *search-condition* follows the same rules used by the search condition in a WHERE clause. In addition, it must not reference any of the following:

- The table for which the column permission is being defined
- A declared global temporary table
- A variable (host variable, SQL variable, SQL parameter, or trigger transition variable)
- A parameter marker
- A user-defined function that is defined as NOT SECURED
- A function that is not deterministic, has an external action, or modifies SQL data
- An OLAP specification
- A ROW CHANGE expression
- A sequence reference
- A * or *name*.* in a select clause
- A table function
- An aggregate function, unless it is specified in a subquery
- A collection-derived table (UNNEST)
- A LOB column or a distinct type column that is based on a LOB
- An XML column
- A view that contains any of the above

ENFORCED FOR ALL ACCESS
Specifies that the row permission applies to all references of the table. If row access control is activated for the table, when the table is referenced in a data manipulation statement, DB2 implicitly applies the row permission to control the access of the table. If the reference of the table is for fetch operation such as SELECT, the application of the row permission determines what set of row can be retrieved by the user who requested the fetch operation. If the reference of the table is for a data change operation such as INSERT, the application of the row permission determines whether all rows to be changed by the user who requested the data change operation.

ENABLE or DISABLE
Specifies that the row permission is to be initially enabled or disabled for row access control.

DISABLE
Specifies that the row permission is to be disabled for row access control. The row permission will remain ineffective regardless of whether row access control is activated for the table or not. This is the default.

ENABLE
Specifies that the row permission is to be enabled for row access control. If row access control is not currently activated for this table, the row permission will become effective when row access control is activated for the table. If row access control is currently activated for the table, the row permission becomes effective immediately.

Notes

How row permissions are applied and how they affect certain statements: See the ALTER TABLE statement with the ACTIVATE ROW ACCESS CONTROL clause for information on how to activate row access control and how row permissions are applied. See the description of subselect for information on how the application

of row permissions affects the fetch operation. See the data change statements for information on how the application of row permissions affects the data change operation.

Row permissions that are created before row access control is activated for a table: The CREATE PERMISSION statement is an independent statement that can be used to create a row permission before row access control is activated for a table. The only requirement is that the table and the columns exist before the permission is created. Multiple row permissions can be created for a table.

The definition of the row permission is stored in the DB2 catalog. Dependency on the table for which the permission is being created and dependencies on other objects referenced in the definition are recorded. A row permission can be created as enabled or disabled for row access control. An enabled row permission does not take effect until the ALTER TABLE statement with the ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for the table. A disabled row permission remains ineffective even when row access control is activated for the table. The ALTER PERMISSION statement can be used to alter between ENABLE and DISABLE.

After row access control is activated for a table, when the table is referenced in a data manipulation statement, all enabled row permissions that are defined for the table are implicitly applied by DB2 to control access to the table.

Creating row permissions before activating row access control for a table is the recommended sequence.

Row permissions that are created after row access control is activated for a table: An enabled row permission becomes effective as soon as it is committed. Thereafter, when the table is referenced in a data manipulation statement, all enabled row permissions are implicitly applied to the statement. Any disabled row permission remains ineffective even when row access control is activated for the table.

No cascaded effect when row or column access control enforced tables are referenced in row permission definitions: A row permission definition may reference tables and columns that are currently enforced by row or column access control. Access control from those tables is ignored when the table for which the row permission is being created is referenced in a data manipulation statement.

Examples

Example 1: Secure user-defined function ACCOUNTING_UDF in row permission SALARY_ROW_ACCESS processes the sensitive data in column SALARY. After row access control is activated for table EMPLOYEE, Accountant Paul retrieves the salary of employee with EMPNO 123456 who is making $100,000 a year. Paul may or may not see the row depending on the output value from user-defined function ACCOUNTING_UDF.

```
CREATE PERMISSION SALARY_ROW_ACCESS ON EMPLOYEE
   FOR ROWS WHERE VERIFY_GROUP_FOR_USER(SESSION_USER,'MGR','ACCOUNTING') = 1
                  AND
                  ACCOUNTING_UDF(SALARY) < 120000
   ENFORCED FOR ALL ACCESS
   ENABLE;

COMMIT;
```

```
ALTER TABLE EMPLOYEE
ACTIVATE ROW ACCESS CONTROL;

COMMIT;

SELECT SALARY FROM EMPLOYEE
   WHERE EMPNO = 123456;
```

Example 2: The tellers in a bank can only access customers from their branch. All tellers have secondary authorization ID TELLER. The customer service representatives are allowed to access all customers of the bank. All customer service representatives have secondary authorization ID CSR. A row permission is created for each group of personnel in the bank accordingly to the access rule defined by SECADM authority. After row access control is activated for table CUSTOMER, in the SELECT statement the search conditions of both row permissions are merged into the statement and they are combined with the logic OR operator to control the set of rows accessible by each group.

```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
   FOR ROWS WHERE VERIFY_GROUP_FOR_USER(SESSION_USER,'TELLER') = 1
                  AND
                  BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
                            WHERE EMP_ID = SESSION_USER)
   ENFORCED FOR ALL ACCESS
   ENABLE;

COMMIT;

CREATE PERMISSION CSR_ROW_ACCESS ON CUSTOMER
   FOR ROWS WHERE VERIFY_GROUP_FOR_USER(SESSION_USER,'CSR') = 1
   ENFORCED FOR ALL ACCESS
   ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
   ACTIVATE ROW ACCESS CONTROL;

COMMIT;

SELECT * FROM CUSTOMER;
```

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 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 667.
- SQL

  The procedure is written exclusively in SQL statements. The body of an SQL procedure is written in the SQL procedural language. The procedure body is defined at the current server along with various attributes of the procedure. See "CREATE PROCEDURE (SQL)" on page 675.

Notes

Defining the parameters: The input parameters for the procedure are specified as a list within parentheses.

The maximum number of parameters allowed in CREATE PROCEDURE is 90, see Chapter 9, "SQL limits," on page 959.

A procedure can have no input parameters. In this case, an empty set of parentheses can be specified or omitted, for example:

```
    CREATE PROCEDURE ASSEMBLY_PARTS()
or
    CREATE PROCEDURE ASSEMBLY_PARTS
```

- **Choosing data types for parameters:** For portability of procedures across platforms, use the following recommended data type names:
  - DOUBLE or REAL instead of FLOAT.
  - DECIMAL instead of NUMERIC.
- **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 data type or a distinct type that is based on a LOB data type.

  AS LOCATOR cannot be specified for SQL procedures.

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.

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

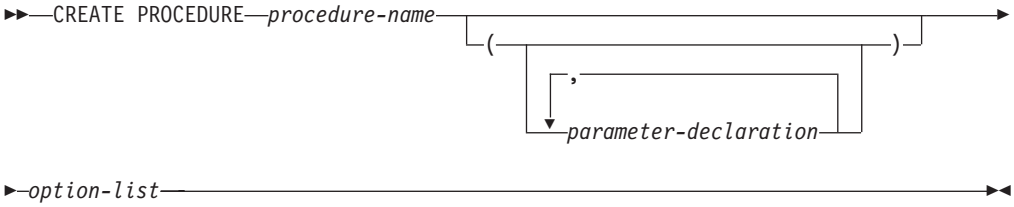
The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema
- Database administrator authority.

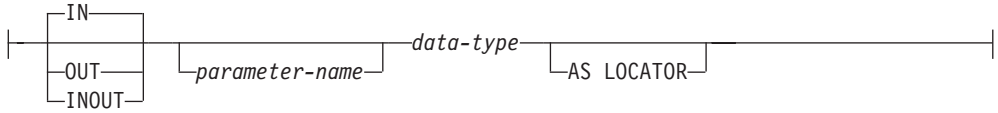
For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Database administrator authority.

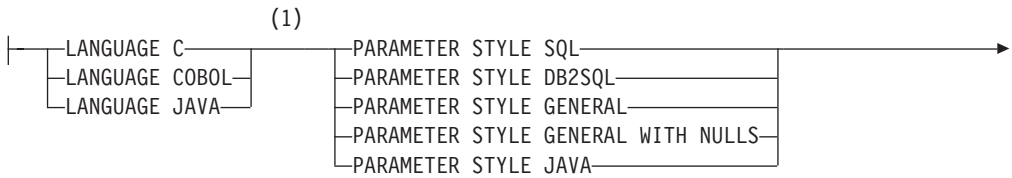
Syntax

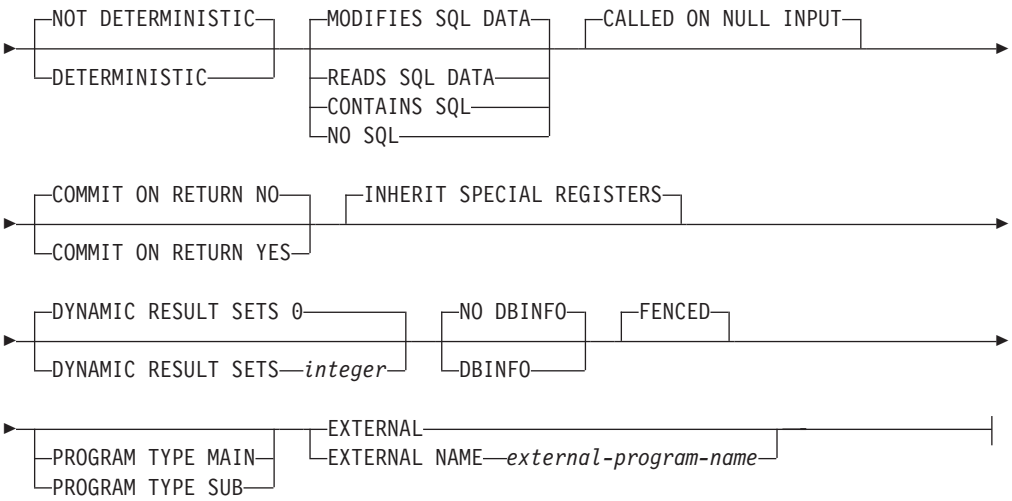


parameter-declaration:



option-list:



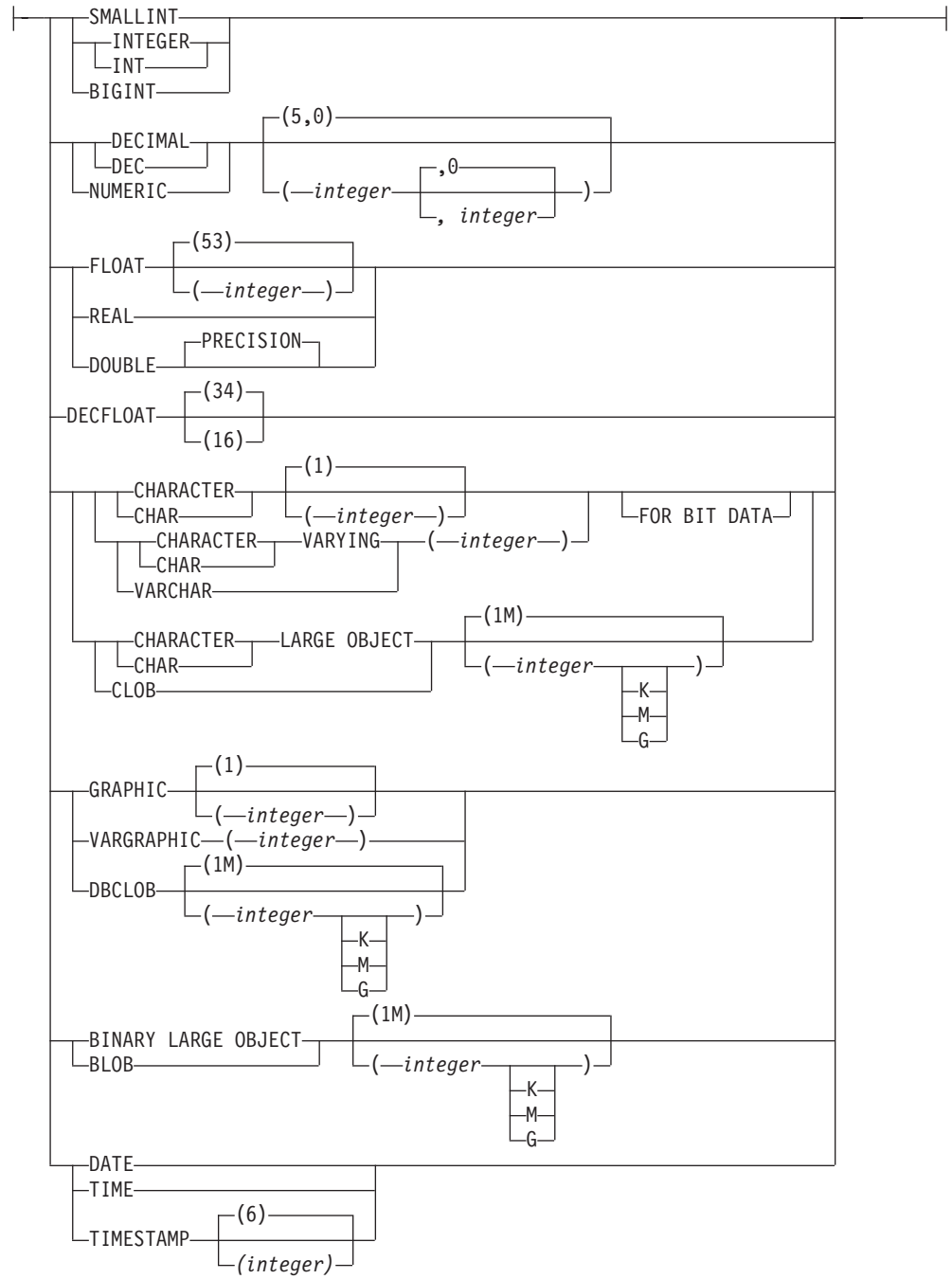


Notes:

1 The clauses in the option-list can be specified in any order.

data-type:

built-in-type:



Description

procedure-name
Names the procedure. The name, including the implicit or explicit qualifier, must not identify a procedure that already exists at the current server. If a qualified procedure name is specified, the *schema-name* must not be one of the system schemas (see "Schemas" on page 11).

(*parameter-declaration***,...)**
Specifies the number of parameters of the procedure, the data type of each parameter, and, optionally, the name of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. All parameters are nullable, with the exception that for Java

procedures, numeric parameters other than DECIMAL and NUMERIC are not nullable. A maximum of 90 parameters can be specified. See Chapter 9, "SQL limits," on page 959 for more details on limits.

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.

INOUT
Identifies the parameter as both an input and output parameter for the procedure.

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. See "CREATE TABLE" on page 688 for a more complete description of each built-in data type.

For parameters with a LOB data type, AS LOCATOR must not be specified when PARAMETER STYLE JAVA is specified.

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 734.

See "Attributes of the arguments of a routine program" on page 1136 for details on the mapping between the SQL data types and host language data types. Some data types are not supported in all languages.

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the procedure instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to and from the procedure, especially when the value of the parameter is very large.

G In DB2 for LUW the AS LOCATOR clause is not supported.

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the procedure body is written. All programs must be designed to run in the server's environment.

C The external program is written in C or C++. The database manager will call the procedure using the C language calling conventions.

COBOL
The external program is written in COBOL. The database manager will call the procedure using the COBOL language calling conventions.

JAVA
The external program is written in Java. The database manager will call the

procedure using the Java language calling conventions. The procedure must be a public static method of the specified Java class.

PARAMETER STYLE

Specifies the conventions used to pass parameters to and return values from procedures. See "Parameter passing for external routines" on page 1127 for details.

SQL

Specifies that in addition to the parameters on the CALL statement, several additional parameters are passed to the procedure. The following arguments are passed to the procedure:

- The first *n* parameters are the parameters that are specified on the CREATE PROCEDURE statement.
- *n* parameters for indicator variables for the parameters.
- The SQLSTATE to be returned.
- The qualified name of the procedure.
- The specific name of the procedure.
- The SQL diagnostic string to be returned.

PARAMETER STYLE SQL cannot be used with LANGUAGE JAVA or with DBINFO.

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:

- If DBINFO is specified, the DBINFO structure.

PARAMETER STYLE DB2SQL cannot be used with LANGUAGE JAVA. You should only consider using this parameter style when DBINFO is also specified.

GENERAL

Specifies that the procedure will use a parameter passing mechanism where the procedure receives the parameters specified on the CALL. Arguments to procedures defined with this parameter style cannot be null.

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.

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 9075-13:2008, *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 used with LANGUAGE JAVA.

NOT DETERMINISTIC or 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. For the classification of each statement, see "SQL statement data access classification for routines" on page 974.

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.

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.

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 32767. The default is DYNAMIC RESULT SETS 0.

NO DBINFO or 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 "Database information in external routines (DBINFO)" on page 1138 for further details.

DBINFO can be specified only if PARAMETER STYLE DB2SQL is specified.

FENCED
Specifies that the procedure runs in an environment that is isolated from the database manager environment.

PROGRAM TYPE
Specifies whether the procedure runs as a main routine or a subroutine. PROGRAM TYPE MAIN is only valid for LANGUAGE C or COBOL.

G The default when PROGRAM TYPE MAIN is not specified is product specific.

SUB
The procedure runs as a subroutine. With LANGUAGE JAVA, PROGRAM TYPE MAIN is not allowed.

MAIN
The procedure runs as a main routine.

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 that will be executed when the procedure is called by the CALL statement. The executable form of the external program need not exist when the CREATE PROCEDURE statement is executed. However, it must exist at the current server when the procedure is called.

If a JAR is referenced then the JAR must exist when the procedure is created.

Notes

General considerations for defining procedures: See "CREATE PROCEDURE" on page 666 for general information on defining procedures.

Language considerations: For information needed to create the programs for a procedure, see Chapter 19, "Coding programs for use by external routines," on page 1127.

Owner privileges: The owner is authorized to call the procedure (EXECUTE) and grant others the privilege to call the procedure. See "GRANT (function or procedure privileges)" on page 793. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

Error handling considerations: 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.

Examples

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 defines 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

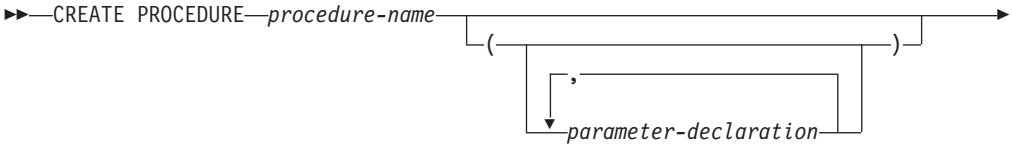
The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema
- Database administrator authority.

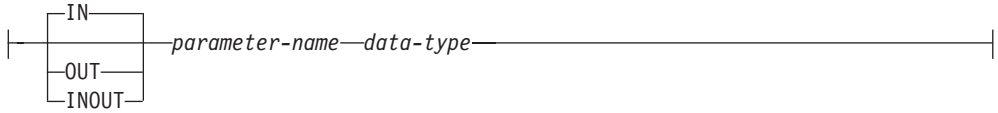
For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Database administrator authority.

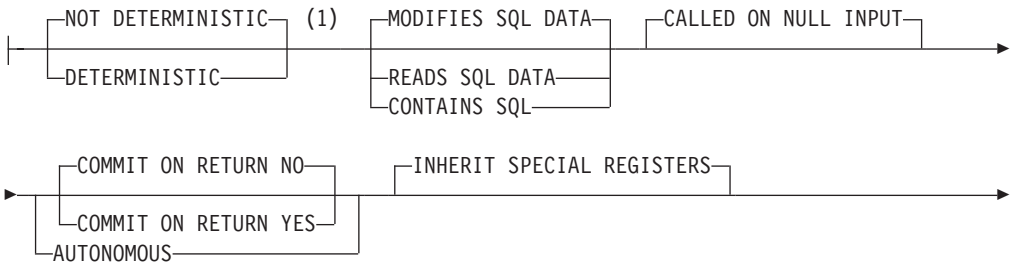
Syntax



parameter-declaration:



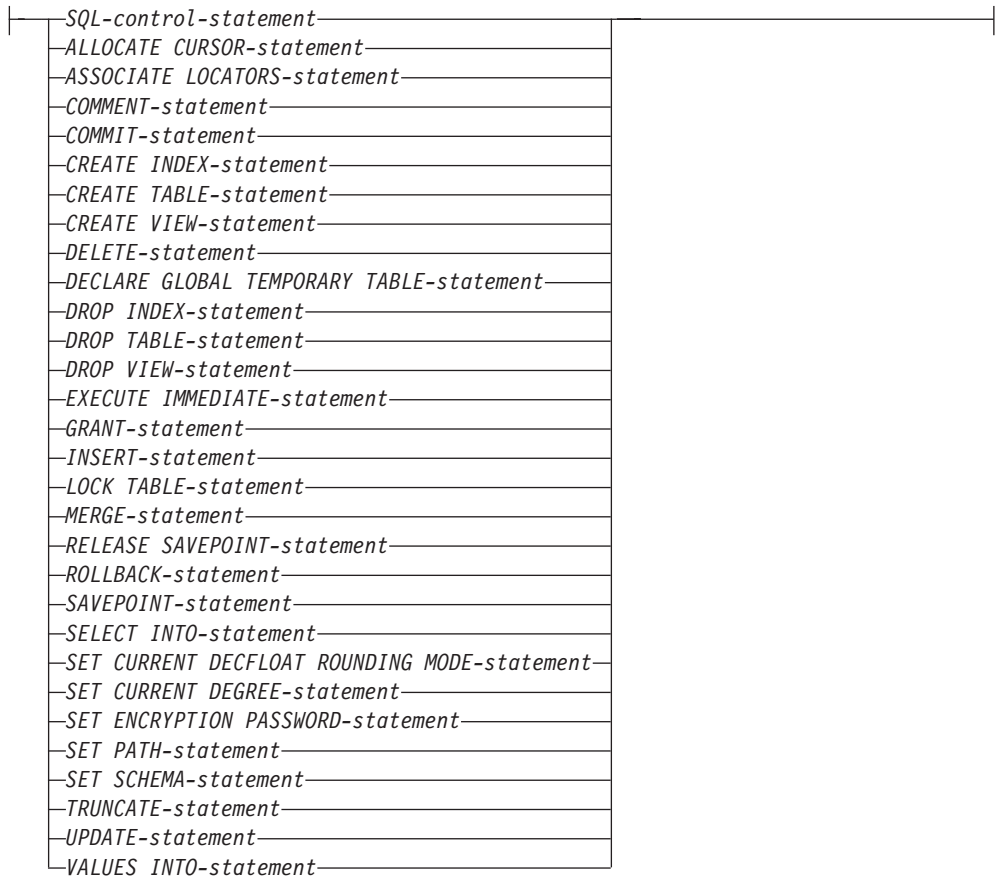
option-list:



Notes:

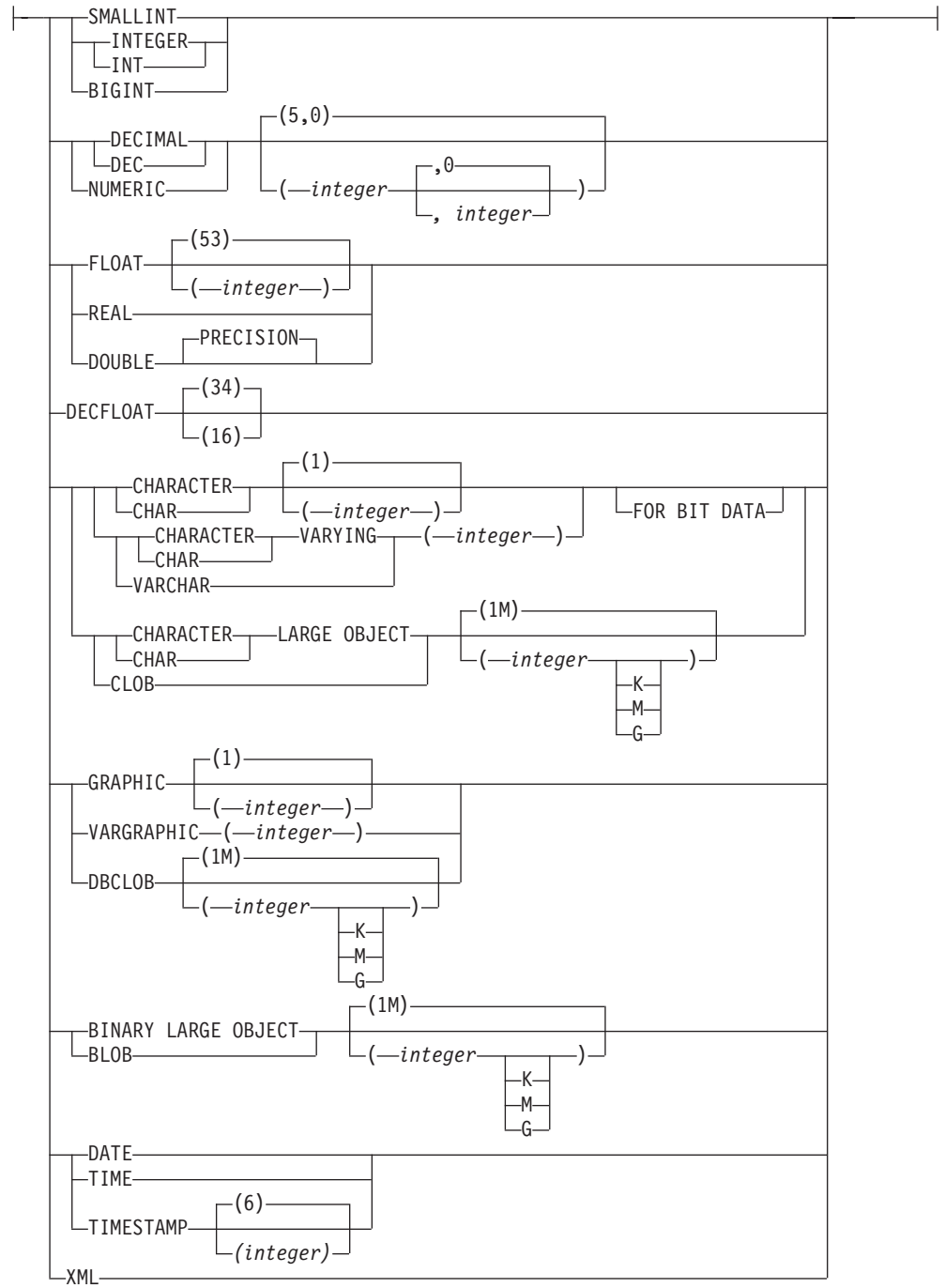
1 The clauses in the option-list can be specified in any order.

SQL-routine-body:



data-type:

built-in-type:



Description

procedure-name

Names the procedure. The name, including the implicit or explicit qualifier, must not identify a procedure that already exists at the current server. If a qualified procedure name is specified, the *schema-name* must not be one of the system schemas (see "Schemas" on page 11).

(*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, of for both input and output. A

maximum of 253 parameters can be specified. All parameters are nullable. See Chapter 9, "SQL limits," on page 959 for more details on limits.

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.

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.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 688 for a more complete description of each built-in data type.

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 734.

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.

See "Attributes of the arguments of a routine program" on page 1136 for details on the mapping between the SQL data types and host language data types. Some data types are not supported in all languages.

LANGUAGE SQL
Specifies that this procedure is written exclusively in SQL.

NOT DETERMINISTIC or 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. For the classification of each statement, see "SQL statement data access classification for routines" on page 974.

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 or CONTAINS SQL.

CONTAINS SQL
Specifies that the procedure can only execute statements with a data access classification of CONTAINS SQL.

CALLED ON NULL INPUT
Specifies that the procedure is called regardless of whether any of the input arguments is null. This specification means that the procedure is responsible for testing for null argument values. The procedure can return null or nonnull values.

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.

AUTONOMOUS
Specifies that the procedure is executed in a unit of work that is independent from the calling application. When this option is specified the database always commits or rolls back the autonomous procedure's transactional work based on the SQLSTATE that is returned from the procedure. A SQLSTATE indication of unqualified success or warning will cause the transaction to be committed. All other SQLSTATEs will cause the autonomous procedure's unit of work to be rolled back.

The invocation of any trigger, function, or procedure from within the autonomous procedure will be part of the autonomous procedure's unit of work unless the trigger, function, or procedure was explicitly created to run under a different activation group.

An autonomous procedure cannot be called directly or indirectly from another autonomous procedure.

If AUTONOMOUS is specified, DYNAMIC RESULT SETS 0 must be specified.

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 32767. The default is DYNAMIC RESULT SETS 0.

SQL-routine-body
Specifies the statements that define the body of the SQL procedure. Multiple SQL procedure statements may be specified within a compound statement or other SQL control statements. See Chapter 8, "SQL control statements," on page 907 for more information.

Notes

General considerations for defining procedures: See "CREATE PROCEDURE" on page 666 for general information on defining procedures.

Owner privileges: The owner is authorized to call the procedure (EXECUTE) and grant others the privilege to call the procedure. See "GRANT (function or procedure privileges)" on page 793. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

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 which correspond to OUT parameters are undefined and those which correspond to INOUT parameters are unchanged when an error is returned by the procedure.

G **References to XML variables and parameters:** In DB2 for LUW, referencing
G variables or parameters of data type XML in SQL procedures after a commit or
G rollback operation occurs, without first assigning new values to these variables, is
G not supported .

Examples

Example 1: 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 SEQUENCE

The CREATE SEQUENCE statement creates a sequence at the application server.

Invocation

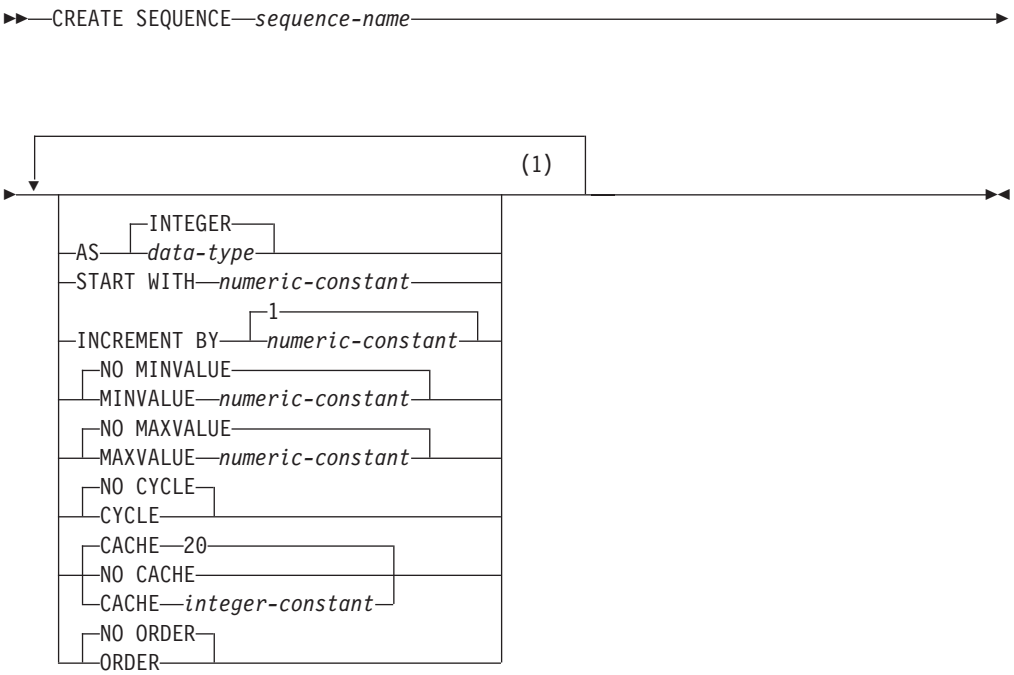
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

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
- Database administrator authority.

Syntax

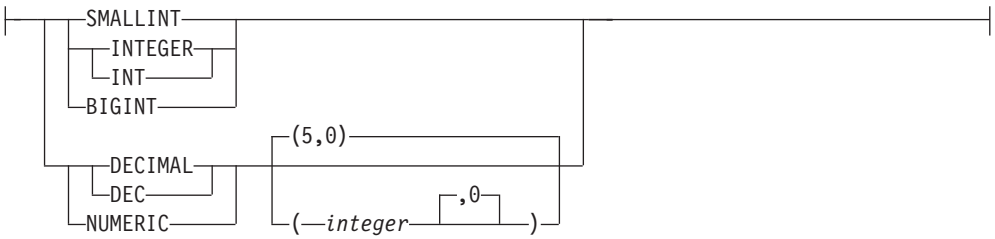


Notes:

1 The same clause must not be specified more than once.

data-type:

built-in-type:



Description

sequence-name
Names the sequence. The name, including the implicit or explicit qualifier, must not identify an existing sequence at the current server. The schema name must not be a system 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, or DECIMAL) 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, the precision must be less than or equal to 31 and the scale must be 0. See "CREATE TABLE" on page 688 for a more complete description of each built-in data type.

distinct-type-name
Specifies that the data type of the sequence is a distinct type. If the source type is DECIMAL, 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 31 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 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 nonzero digits existing to the right of the decimal point.

If the START WITH clause is not explicitly specified with a value, the default is MINVALUE for ascending sequences and MAXVALUE for descending sequences.

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. The value must not exceed the value of a large integer constant and must not contain nonzero digits 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 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 nonzero digits existing 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 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 nonzero digits existing 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 the sequence should continue to generate values after reaching either its maximum or minimum value. 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 its maximum value it generates its minimum value; or after a descending sequence reaches its minimum value 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, 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 a 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 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 the case of a system failure, shutdown or database deactivation. 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 a synchronous write.

NO ORDER or ORDER
Specifies whether the sequence numbers 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

Owner privileges: The owner always acquires the ALTER and USAGE privilege on the sequence with the ability to grant these privileges to others.

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 (that is, CYCLE was explicitly specified), the maximum or minimum value generated for a sequence might 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, carefully consider the impact of the values for MINVALUE, MAXVALUE and START WITH.

Caching sequence numbers: Caching sequence numbers implies that 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 may require having to wait.

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 may require having to wait. 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 or altered, or until the end of the SQL-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 prior to 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.

Examples

Example 1: Create a sequence called ORDER_SEQ that starts at 1, increments by 1, does not cycle, and caches 24 values at a time:

```
CREATE SEQUENCE ORDER_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 the default, had they 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 its 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
- Database administrator authority.

If defining 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 on the table
- The REFERENCES privilege on each column of the specified parent key
- Ownership of the table
- Database administrator authority.

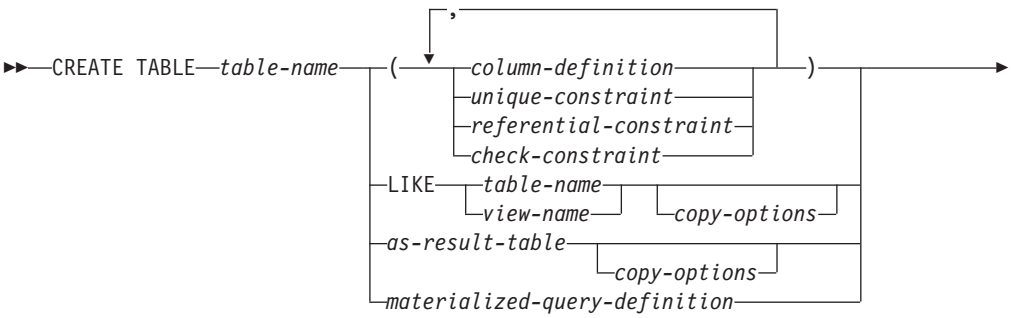
If the LIKE clause is specified, the privileges held by the authorization ID of the statement must include at least one of the following on the table or view specified in the LIKE clause:

- The SELECT privilege for the table or view
- Ownership of the table or view
- Database administrator authority.

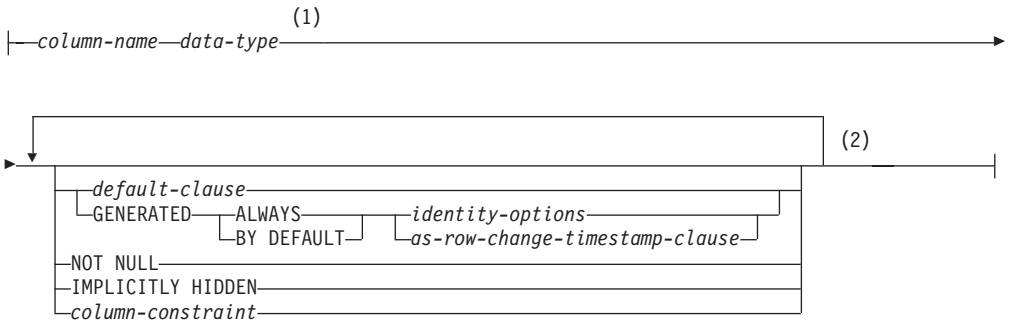
If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- The USAGE privilege on the distinct type
- Ownership of the distinct type
- Database administrator authority.

Syntax



column-definition:



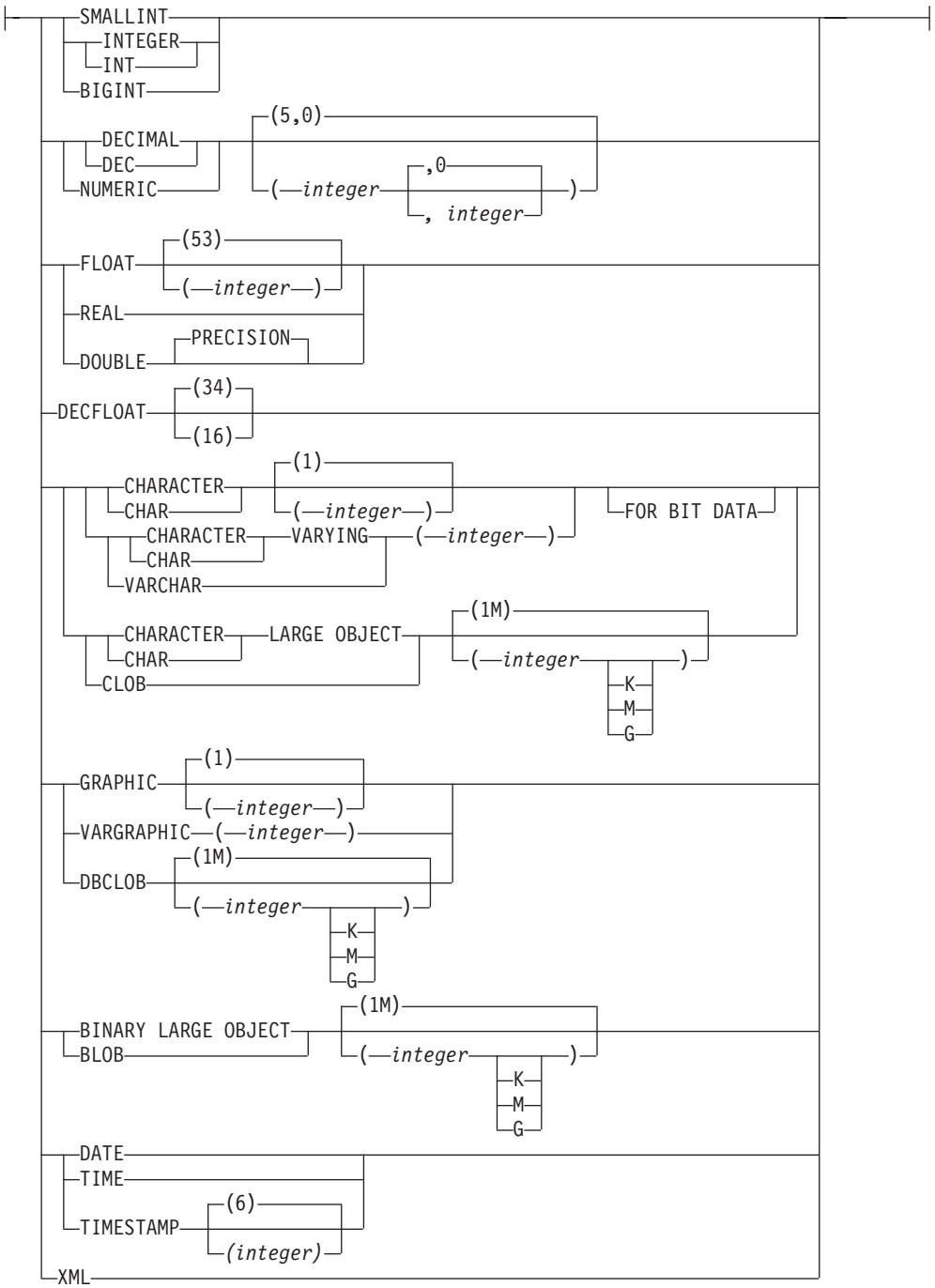
Notes:

1 *data-type* is optional if *as-row-change-timestamp-clause* is specified

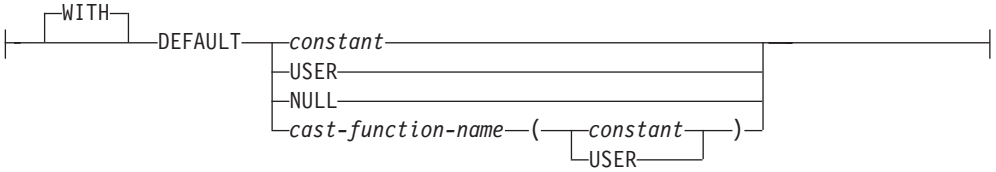
2 The same clause must not be specified more than once.

data-type:

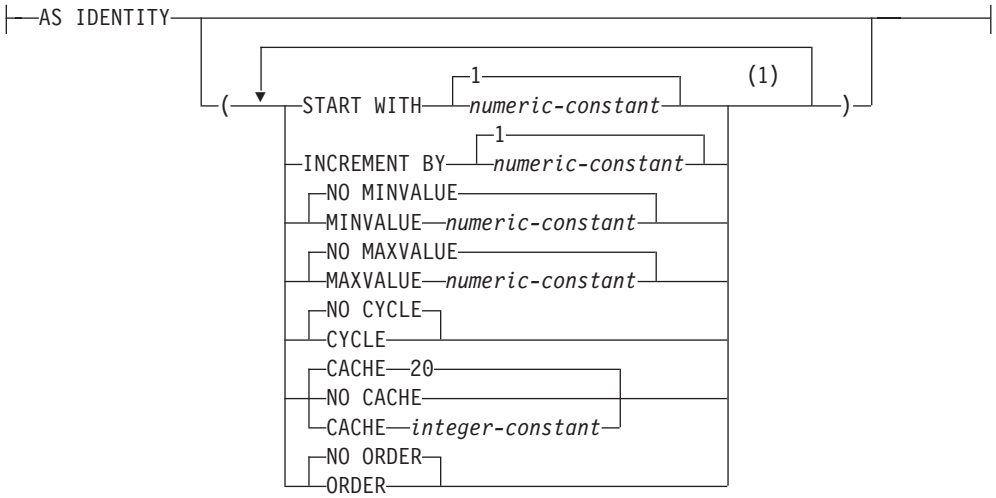
built-in-type:



default-clause:



identity-options:

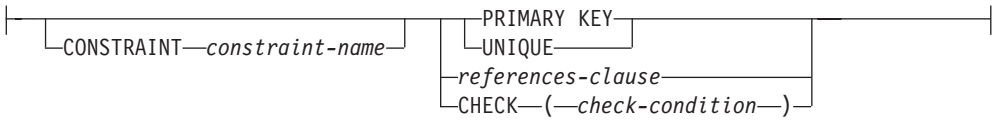


as-row-change-timestamp-clause:

Notes:

1 The same clause must not be specified more than once.

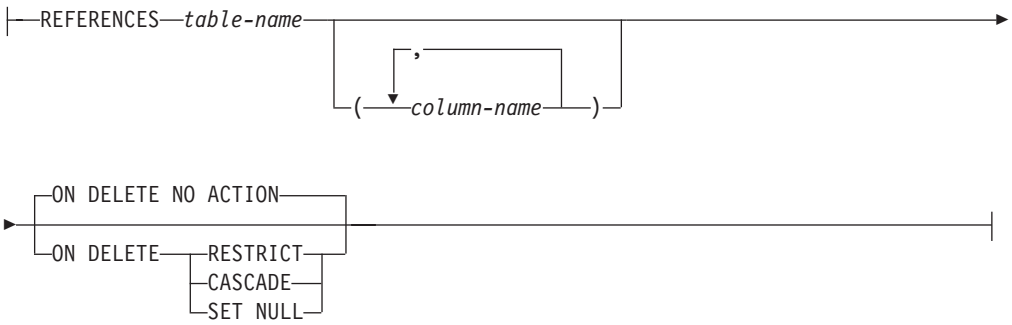
column-constraint:



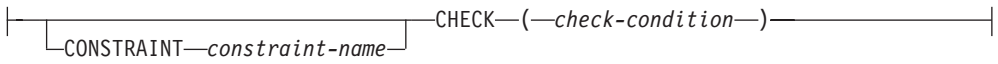
unique-constraint:

referential-constraint:

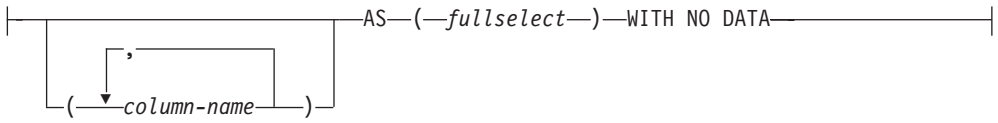
references-clause:



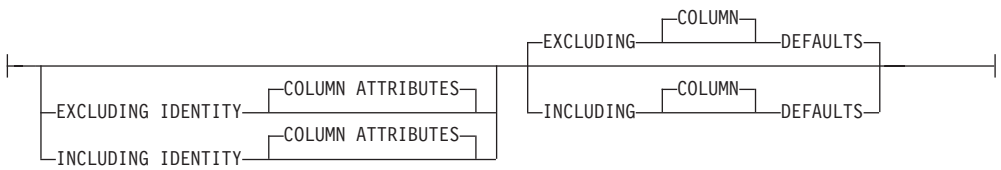
check-constraint:



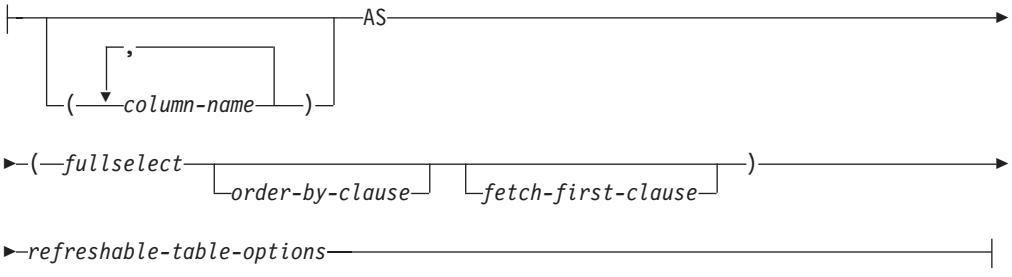
as-result-table:



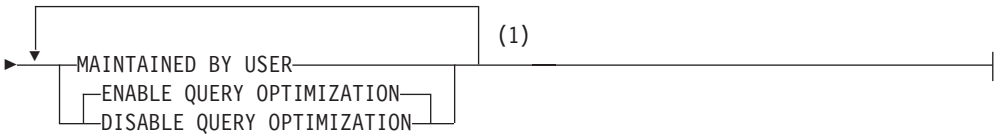
copy-options:



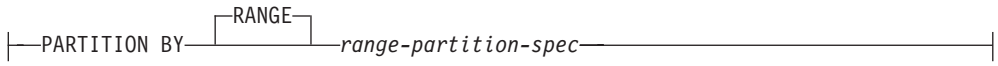
materialized-query-definition:



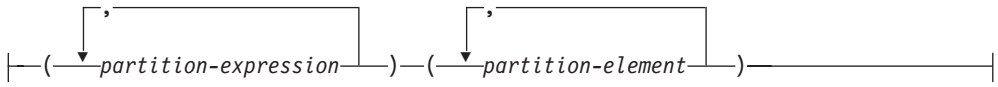
refreshable-table-options:



partitioning-clause:



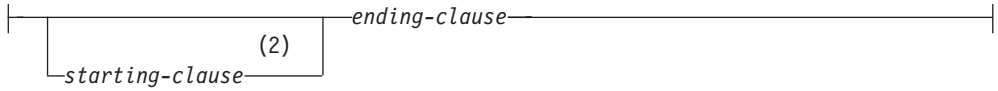
range-partition-spec:



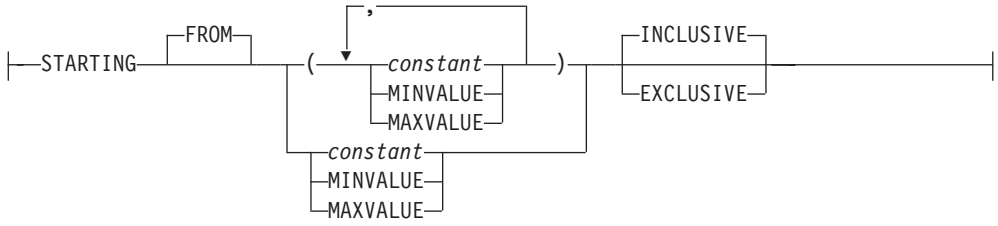
partition-expression:

partition-element:

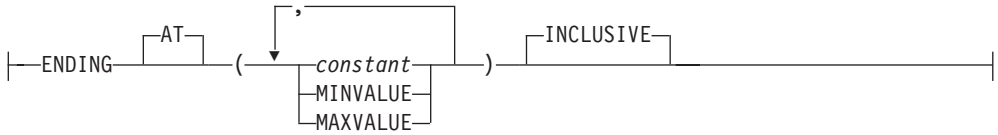
boundary-spec:



starting-clause:



ending-clause:



Notes:

1 The same clause must not be specified more than once. MAINTAINED BY USER must be specified.

G 2 In DB2 for z/OS, the *starting-clause* must not be specified. In DB2 for i, the
G *starting-clause* is optional. In DB2 for LUW, the first *partition-element* must
G include a *starting-clause*.

Description

table-name
Names the table. The name, including the implicit or explicit qualifier, must not identify an alias, index, table or view that already exists at the current server.

column-definition

Defines the attributes of a column. There must be at least one column definition and no more than 750 [113] columns for the table. See Table 63 on page 967 for more information.

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.

data-type
Specifies the data type of the column.

built-in-type
For the 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 31. The second integer is the scale of the number (the number of digits to the right of the decimal point). The scale of the number 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***)**
NUMERIC(*integer***)**
NUMERIC
G For a zoned decimal number in DB2 for i and a packed decimal number in
G DB2 for z/OS, and DB2 for LUW. The first integer is the precision of the number; that is, the total number of digits; it can range from 1 to 31. The

113. This value is 1 less if the table is a dependent table.

second integer is the scale of the number (the number of digits to the right of the decimal point). The scale of the number can 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(*integer***)**
FLOAT
For a single- or double-precision floating-point number, depending on the value of the integer. The value of the integer must be in the range 1 through 53. The values 1 through *n* indicate single-precision, and the values *n* + 1 through 53 indicate double-precision. In DB2 for z/OS, *n* is 21; in DB2 for LUW and DB2 for i, *n* is 24. For portability across operating systems, when specifying a floating-point data type, use REAL or DOUBLE PRECISION instead of FLOAT.

You can use DOUBLE PRECISION or FLOAT for FLOAT(53).

REAL
For a single-precision floating-point number.

DOUBLE PRECISION or DOUBLE
For a double-precision floating-point number.

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 to 254. See Table 60 on page 964 for more information.

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. The integer can range from 1 to 32 672. See Table 60 on page 964 for more information.

FOR BIT DATA
Indicates that the values of the CHAR or VARCHAR column are not associated with a coded character set and therefore are never converted. The CCSID of a bit data column is X'FFFF'. In DB2 for LUW, the CCSID for a bit data column is X'0000'.

If this clause is omitted, the CCSID of a SBCS or mixed data column is the corresponding default CCSID at the current server.

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. A CLOB column has a varying length. It cannot be referenced in certain contexts regardless of its maximum length. For details, see “Limitations on use of strings” on page 64.

If the length specification is omitted, a length of 1M bytes is assumed.

G To create LOB columns greater than 1 gigabyte in DB2 for LUW, there are
G additional requirements. See product documentation.

The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

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 is 1024 times *integer*.

integer **M**
The maximum value for *integer* is 2048. The maximum length is 1 048 576 times *integer*.

integer **G**
The maximum value for *integer* is 2. The maximum length is 1 073 741 824 times *integer*.

If a value that evaluates to 2 gigabytes (2 147 483 648) is specified, then the value that is actually used is one byte less, that is 2 147 483 647.

GRAPHIC(*integer***)**
GRAPHIC
For a fixed-length graphic string of length *integer*. The integer can range from 1 to 127. See Table 60 on page 964 for more information.

If the length specification is omitted, a length of 1 is assumed.

VARGRAPHIC(*integer***)**
For a varying-length graphic string of maximum length *integer*. The integer can range from 1 to 16 336. See Table 60 on page 964 for more information.

DBCLOB(*integer***[K|M|G])**
DBCLOB
For a double-byte character large object string of the specified maximum length in double-byte characters. The maximum length must be in the range of 1 through 1 073 741 823. A DBCLOB column has a varying length. It cannot be referenced in certain contexts regardless of its maximum length. For details, see “Limitations on use of strings” on page 64.

If the length specification is omitted, a length of 1M double-byte characters is assumed.

G To create LOB columns greater than 1 gigabyte in DB2 for LUW, there are
G additional requirements. See product documentation.

The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

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 1 048 576. The maximum length is 1024 times *integer*.

integer **M**
The maximum value for *integer* is 1024. The maximum length is 1 048 576 times *integer*.

integer **G**
The maximum value for *integer* is 1. The maximum length is 1 073 741 824 times *integer*.

If a value that evaluates to 2 gigabytes (1 073 741 824 double-byte characters) is specified, then the value that is actually used is one double-byte character less, that is 1 073 741 823.

BINARY LARGE OBJECT(*integer* **[K|M|G]) or BLOB(***integer* **[K|M|G])**
BINARY LARGE OBJECT or BLOB
For a binary 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. A BLOB column has a varying length. It cannot be referenced in certain contexts regardless of its maximum length. For details, see "Limitations on use of strings" on page 64.

If the length specification is omitted, a length of 1M bytes is assumed.

G To create LOB columns greater than 1 gigabyte in DB2 for LUW, there are
G additional requirements. See product documentation.

The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

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 is 1024 times *integer*.

integer **M**
The maximum value for *integer* is 2048. The maximum length is 1 048 576 times *integer*.

integer **G**
The maximum value for *integer* is 2. The maximum length is 1 073 741 824 times *integer*.

If a value that evaluates to 2 gigabytes (2 147 483 648) is specified, then the value that is actually used is one byte less, that is 2 147 483 647.

DATE
For a date.

TIME
For a time.

| **TIMESTAMP(***integer***) or TIMESTAMP**
| For a timestamp. The integer must be between 0 and 12 and specifies the
| precision of fractional seconds from 0 (seconds) to 12 (picoseconds). The
| default is 6 (microseconds).

XML
For an XML document. Only well-formed documents can be inserted into an XML column.

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 partitioning clause of a partitioned table.

distinct-type-name
Specifies 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 in the SQL path.

DEFAULT
Specifies a default value for the column. This clause must not be specified more than once in the same *column-definition*. DEFAULT cannot be specified for the following types of columns :

- An identity column (the database manager generates default values)
- A row change timestamp column (the database manager generates default values)
- An XML column

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 77. A 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 an SQL data change statement as the default for the column. The data type of the column or 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 the USER special register.

NULL
Specifies null as the default for the column. If NOT NULL was specified, DEFAULT NULL must not be specified within the same *column-definition*.

cast-function-name
The name of the cast function that matches the name of the distinct type name of the data type for the column.

The schema name of the cast function, whether it is explicitly specified or implicitly resolved through function resolution, must be the same as the explicitly or implicitly specified schema name of the distinct type. This form of the DEFAULT value can only be used with columns that are defined as a 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.

USER
Specifies the value of the USER special register at the time of an SQL data change statement as the default for the column. The source type of the distinct type of the column must be a CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the USER special register.

If the value specified is not valid, an error is returned.

GENERATED
Specifies that the database manager generates values for the column. GENERATED must 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 the identity column for the table. A table can only have a single 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, or a distinct type for which the source type is an exact numeric type with a scale of zero.

An identity column is implicitly NOT NULL. An identity column cannot have a DEFAULT clause.

Defining a column AS IDENTITY does not necessarily guarantee uniqueness of the values. To ensure uniqueness of the values, define a unique, single-column index on the identity column.

START WITH *numeric-constant*
Specifies the first value that is generated for the identity column. This value can be any positive or negative value that could be assigned to this 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 identity column and the MAXVALUE for a descending identity column. This value is not necessarily the value that would be cycled to after reaching the maximum or minimum value for the identity column. The START WITH clause can be used to start the generation of values 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. This value can be any positive or negative value, including zero, that could be assigned to this column, and does not exceed the value of a large integer constant, without non-zero digits existing to the right of the decimal point.

If this value is negative, this is a descending identity column. If this value is 0, or positive, this is an ascending identity column. The default is 1.

NO MINVALUE or MINVALUE
Specifies the minimum value at which a descending identity column either cycles or stops generating values, or an ascending identity column cycles to after reaching the maximum value.

NO MINVALUE
For an ascending sequence, the value is the START WITH value, or 1 if START WITH was not specified. For a descending sequence, the value is the minimum value of the data type of the column. This is the default.

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, without non-zero digits existing to the right of the decimal point, but the value must be less than or equal to the maximum value.

NO MAXVALUE or MAXVALUE
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.

NO MAXVALUE
For an ascending sequence, the value is the maximum value of the data type of the column. For a descending sequence, the value is the START WITH value, or -1 if START WITH was not specified. This is the default.

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, without non-zero digits existing to the right of the decimal point, but the value must be greater than or equal to the minimum value.

NO CYCLE or CYCLE
Specifies whether this identity column should continue to generate values after generating either its maximum or minimum value.

NO CYCLE
Specifies that values will not be generated for the identity column once the maximum or minimum value has been reached. This is the default.

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 identity column reaches the maximum value, it generates its minimum value; or after a descending identity column

reaches the minimum value, it generates its maximum value. The maximum and minimum values for the identity 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, an error is returned.

NO CACHE or 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. The default is CACHE 20.

NO CACHE
Specifies that values for the identity column are not preallocated.

CACHE *integer-constant*
Specifies the maximum number of values for the identity column that can be preallocated by the database manager and kept in memory.

During a failure, all cached identity column values that are yet to be assigned might be lost and will not be used. Therefore, the value specified for CACHE also represents the maximum number of values for the identity column that could be lost during a failure.

The minimum value is 2.

NO ORDER or ORDER
Specifies whether the identity values must be 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.

ORDER
Specifies that the values are generated in order of request.

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 with a precision of 6 or a distinct type based on a TIMESTAMP with a precision of 6. 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. NOT NULL is required for a row change timestamp column.

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*
Names the constraint. A *constraint-name* must not be the same as a constraint name 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.[114] 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.

The NOT NULL clause must be specified with this 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. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

UNIQUE
Provides a shorthand method of defining a unique constraint composed of a single column. [114] 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.

The NOT NULL clause must be specified with this clause. This clause cannot be specified more than once in a column definition and must not be specified if the PRIMARY KEY clause is specified in the column definition. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

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. Columns with the following data types cannot be specified:

114. In DB2 for z/OS, the table is marked as unavailable until all the required indexes are explicitly created, unless the CREATE TABLE statement is processed by the schema processor or the table space that contains the table is implicitly created.

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML
- A row change timestamp

For more information, see REFERENCES clause.

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. For more information, see CHECK clause.

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 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, and that column must be defined as NOT NULL.

The same column must not be identified more than once. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

The number of identified columns must not exceed 64 and the sum of their byte count (see Byte Counts) must not exceed 2000. See Table 63 on page 967 for more information.

A unique index on the identified columns is created during the execution of the CREATE TABLE statement and this index is designated as the primary index of the table.[114]

UNIQUE(*column-name***,...)**
Defines a unique constraint composed of the identified columns.[114] The UNIQUE clause can be specified more than once. Do not identify columns that are 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, and that column must be defined as NOT NULL.

The same column must not be identified more than once. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

The number of identified columns must not exceed 64 and the sum of their byte count (see Byte Counts) must not exceed 2000. See Table 63 on page 967 for more information.

A unique index on the identified columns is created during the execution of the CREATE TABLE statement.[114]

referential-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 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.

(*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. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML
- A row change timestamp

The number of identified columns must not exceed 64 and the sum of their byte count (see Byte Counts) must not exceed 2000. See Table 63 on page 967 for more information.

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 current server, but it must not identify a catalog table or a declared temporary table.

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. In DB2 for z/OS, duplicate referential constraints are ignored with a warning.

Let T2 denote the identified parent table and let T1 denote the table being created. For DB2 for z/OS, T2 must not be the table being created except when the statement is processed by the schema processor.

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 and other attributes.

If a foreign key column is a distinct type, the data type of the corresponding column of the parent key must have the same distinct type.

(*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 data type of the column must not be a LOB data type or a distinct type based on a LOB data type or a row change timestamp. The number of identified columns must not exceed 64 and the sum of their byte count (see Byte Counts) must not exceed 2000. See Table 63 on page 967 for more information.

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. 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 four possible actions:

- NO ACTION (default) [115]
- RESTRICT
- CASCADE
- SET NULL

SET NULL must not be specified unless some column of the foreign key allows null values.

In DB2 for LUW, a self-referencing table with a SET NULL or RESTRICT rule must not be a dependent in a referential constraint with a delete rule of CASCADE.

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 is returned and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.

115. In DB2 for z/OS, the default depends on the value of the CURRENT RULES special register when the CREATE TABLE statement is processed. If the value of the register is 'DB2', the default is RESTRICT. If the value is 'SQL', the default is NO ACTION.

- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

A cycle involving two or more tables must not cause a table to be delete-connected to itself unless all of the delete rules in the cycle are CASCADE. Thus, if the relationship would form a cycle and T2 is already delete-connected to T1, then the constraint can only be defined if it has a delete rule of CASCADE and all other delete rules of the cycle are CASCADE.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. Let T3 denote a table identified in another FOREIGN KEY clause (if any) of the CREATE TABLE statement. The delete rules of the relationships involving T2 and T3 must be the same and must not be SET NULL if:

- T2 and T3 are the same table.
- T3 is a descendant of T2 and the deletion of rows from T2 cascades to T3.
- T2 is a descendant of T3 and the deletion of rows from T3 cascades to T2.
- T2 and T3 are both descendants of the same table and the deletion of rows from that table cascades to both T2 and T3.

If *r* is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as *r*.

check-constraint

CONSTRAINT *constraint-name*

Names the check constraint. A *constraint-name* must not be the same as a constraint name 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. [116]

The *check-condition* is a form of the *search-condition*, except:

- Columns with the following data types cannot be specified:
  - DECFLOAT or a distinct type based on a DECFLOAT
  - A LOB data type or a distinct type based on a LOB data type
  - XML
- It can be up to 3800 bytes long, not including redundant blanks. See Table 63 on page 967 for more information.
- It must not contain any of the following:
  - Subqueries
  - Built-in functions
  - Variables

116. In DB2 for z/OS, the value of the CURRENT RULES special register must be 'STD' to get this behavior.

- Global variables
- Parameter markers
- *sequence-reference*s
- OLAP specifications
- Special registers
- User-defined functions (except cast functions generated for distinct types)
- CASE expressions

In DB2 for z/OS, the *check-condition* is subject to additional restrictions. See the product reference for further information.

For more information about *search-condition*, see "Search conditions" on page 178.

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 each of the columns of *table-name*, or result columns of *view-name* (if applicable to the data type).

- Column name
- Data type, length, precision and scale
- CCSID
- Nullability

For base tables, the default value, identity, row change timestamp, and hidden attribute are also included in the table definition. For a view, if the column of the underlying base table has a default value, then the effect is product-specific.

The implicit definition does not include any other attributes of the identified table or view. For example, the new table does not have any primary key or foreign key.

copy-options

These options specify whether or not to copy additional attributes of the result table definition (table, view, or fullselect).

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 attributes, if any, of the columns resulting from the *fullselect*, *table-name* or *view-name*. In general, the identity attributes are copied if the element of the corresponding column in the *fullselect*, table or view 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 that is an identity column. If the INCLUDING IDENTITY

COLUMN ATTRIBUTES clause is specified with the AS *fullselect* clause, the columns of the new table do not inherit the identity attribute in the following cases:

- The select list of the *fullselect* includes multiple instances of the name of an identity column (that is, selecting the same column more than once).
- The select list of the *fullselect* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *fullselect* includes a set operation (union).

If INCLUDING IDENTITY is not specified, the table will not have an identity column.

If the LIKE clause identifies a view, INCLUDING IDENTITY COLUMN ATTRIBUTES must not be specified.

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
Specifies whether column defaults are inherited.

EXCLUDING COLUMN DEFAULTS
Specifies that the column defaults are not inherited from the *fullselect*, *table-name* or *view-name*. 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 column defaults for each updatable column of the definition of the source table are inherited. Columns that are not updatable do not have a default defined in the corresponding column of the created table.

as-result-table

column-name
Names the columns 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 *fullselect*. 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 *fullselect*.

A list of column names must be specified if the result table of the *fullselect* has duplicate column names or an unnamed column. An unnamed column is a column derived from a constant, function, expression, or set operation that is not named using the AS clause.

fullselect
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 *fullselect* if the *fullselect* were to be executed. The use of AS (*fullselect*) is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *fullselect*.

The implicit definition includes the following attributes of the *n* columns (if applicable to the data type):

- Column name

- Data type, length, precision and scale
- CCSID
- Nullability

The following attributes are not included (the default value and identity 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 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 *fullselect* must not:

- Result in a column with the following data types:
  - A LOB data type or a distinct type based on a LOB data type
  - XML
- Contain PREVIOUS VALUE or a NEXT VALUE expression.
- Refer to variables or include parameter markers

WITH NO DATA
Specifies that the query is used only to define the attributes of the new a table. The table is not populated using the results of the query and the REFRESH TABLE statement cannot be used. When the WITH NO DATA clause is specified, the table is not considered a materialized query table.

materialized-query-definition

column-name
Names the columns 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 *fullselect*. 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 *fullselect*.

A list of column names must be specified if the result table of the *fullselect* has duplicate column names or an unnamed column. An unnamed column is a column derived from a constant, function, expression, or set operation that is not named using the AS clause.

fullselect
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 *fullselect* if the *fullselect* were to be executed. The use of AS (*fullselect*) is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *fullselect*.

The implicit definition includes the following attributes of the *n* columns (if applicable to the data type):

- Column name
- Data type, length, precision and scale
- CCSID
- Nullability

The following attributes are not included (the default value and identity 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 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 *fullselect* must not:

- Result in a column with the following data types:
  - A LOB data type or a distinct type based on a LOB data type
  - XML
- Contain PREVIOUS VALUE or a NEXT VALUE expression.
- Refer to variables, global variables, or include parameter markers

order-by-clause

Specifies the ordering of the rows of the result table of the *fullselect*. See "order-by-clause" on page 494.

fetch-first-clause

Specifies the maximum number of rows that can be retrieved by the *fullselect*. See "fetch-first-clause" on page 497.

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 *fullselect*.

A materialized query table whose *fullselect* contains a GROUP BY clause is summarizing data from the tables referenced in the *fullselect*. 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 *fullselect* restrictions apply:

- The *fullselect* cannot contain a reference to another materialized query table or to a view that refers to a materialized query table.
- The *fullselect* cannot contain a reference to a declared global temporary table in the FROM clause.
- The *fullselect* cannot contain a *data-change-table-reference*.
- The *fullselect* cannot contain a reference to a view that references another materialized query table or declared global temporary table. When a materialized query table is defined with ENABLE QUERY OPTIMIZATION, the *fullselect* cannot contain a reference to a view that contains one of the restrictions from the following paragraph.

When a materialized query table is defined with ENABLE QUERY OPTIMIZATION, the following additional *fullselect* restrictions apply:

- Must be a subselect.

- Must not include any non-deterministic functions or functions that have external actions. For example, a user-defined function that is defined with either EXTERNAL ACTION or NOT DETERMINISTIC or the RAND built-in function cannot be referenced.
- Must not contain any predicates that include subqueries.
- Must not contain:
  - A nested table expression or view that requires temporary materialization
  - A join using the INNER JOIN syntax
  - An outer join
  - A special register
  - A scalar fullselect
  - A row change timestamp column
- If the subselect references a view, the *fullselect* 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.

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 *fullselect* 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.

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. See table 46 on page 674 for more information.

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 of 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. Columns with the following data types cannot be specified:

- DECFLOAT or a distinct type based on a DECFLOAT
- A LOB data type or a distinct type based on a LOB data type
- XML

The number of identified columns must not exceed 16. The sum of length attributes of the columns must not be greater than 255 - n, where n is the number of columns that can contain null values.

NULLS LAST
Indicates that null values compare high.

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. In DB2 for z/OS the name might be further restricted based on the physical attributes for the table. See product documentation for details.

boundary-spec
Specifies the boundaries of a range partition. The boundaries must be specified in ascending sequence. The specified ranges must not overlap. In DB2 for z/OS, if the concatenation of all the values specified for a boundary of a range partition exceeds 255 bytes, only the first 255 bytes are considered.

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. In DB2 for z/OS, the *starting-clause* must not be specified. In DB2 for i, the *starting-clause* is optional. In DB2 for LUW, the first *partition-element* must include a *starting-clause*.

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 *starting-clause* must also be MAXVALUE.

INCLUSIVE
Indicates that the specified range values are included in the data partition.

EXCLUSIVE
Indicates that the specified constant 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.

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.

Notes

Owner privileges: The owner of the table has all table privileges (see "GRANT (table or view privileges)" on page 801) with the ability to grant these privileges to others.

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.

When a table is recovered to a point-in-time, 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.

G **Creating referential constraints:** On DB2 for LUW, the creation of referential
G constraints may invalidate access plans.

Creating a materialized query table: You should create the materialized query table 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, 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.

G **Names of indexes or constraints created automatically:** When an index or
G constraint is automatically created during execution of the CREATE TABLE
G statement, the method used to generate the name is product specific.

CCSIDs for character and graphic columns: The CCSID of a SBCS, graphic, or mixed data column is the corresponding default CCSID at the current server.

Byte counts: The sum of the byte counts of the columns must not be greater than 32 677. See Table 63 on page 967 for more information.

The following table contains the byte counts of columns by data type for columns that do not allow null values. In DB2 for z/OS and DB2 for LUW, a column that allows null values has a byte count that is one more than shown in the list. In DB2 for i, if any column allows null values, one byte is required for every eight columns.

Data Type	Byte Count
SMALLINT	2
INTEGER	4
BIGINT	8
DECIMAL(*p*,*s*)	the integral part of (*p*/2) + 1
NUMERIC(*p*,*s*)	In DB2 for z/OS and DB2 for LUW, the integral part of (p/2)+1. In DB2 for i, p.
FLOAT (single precision)	4
FLOAT (double precision)	8
DECFLOAT(16)	8
DECFLOAT(34)	16
CHAR(*n*)	*n*
VARCHAR(*n*)	In DB2 for z/OS and DB2 for i, n+2. In DB2 for LUW, n+4.
CLOB(*n*)	Product-specific.
GRAPHIC(*n*)	2*n*
VARGRAPHIC(*n*)	In DB2 for z/OS and DB2 for i, (n*2)+2. In DB2 for LUW, (n*2)+4.
DBCLOB(*n*)	Product-specific.
BLOB(*n*)	Product-specific.
DATE	4
TIME	3
TIMESTAMP(*p*)	The integral part of ((p+1)/2) + 7
XML	Product-specific.
distinct type	The byte count for the source data type.

Examples

Example 1: Given database administrator 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 a table named ACT, which has an identity column named ACTNO. Define the identity column so that the database manager will generate the values for the column by default. Start the values at 10 and increment by 10. Make the identity column unique so that if a value is explicitly assigned to the identity column, it does not duplicate existing values.

```
CREATE TABLE ACT
  (ACTNO     SMALLINT    NOT NULL
     GENERATED BY DEFAULT AS IDENTITY
       (START WITH 10
        INCREMENT BY 10),
   ACTKWD    CHAR(6)     NOT NULL,
   ACTDESC   VARCHAR(20) NOT NULL,
   UNIQUE(ACTNO))
```

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
- Database administrator authority.

The privileges held by the authorization ID of the statement must include at least one of the following:

- Each of the following:
  - 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 *triggered-action search-condition*, and
  - The privileges required to execute each *triggered-SQL-statement*.
- Database administrator authority.

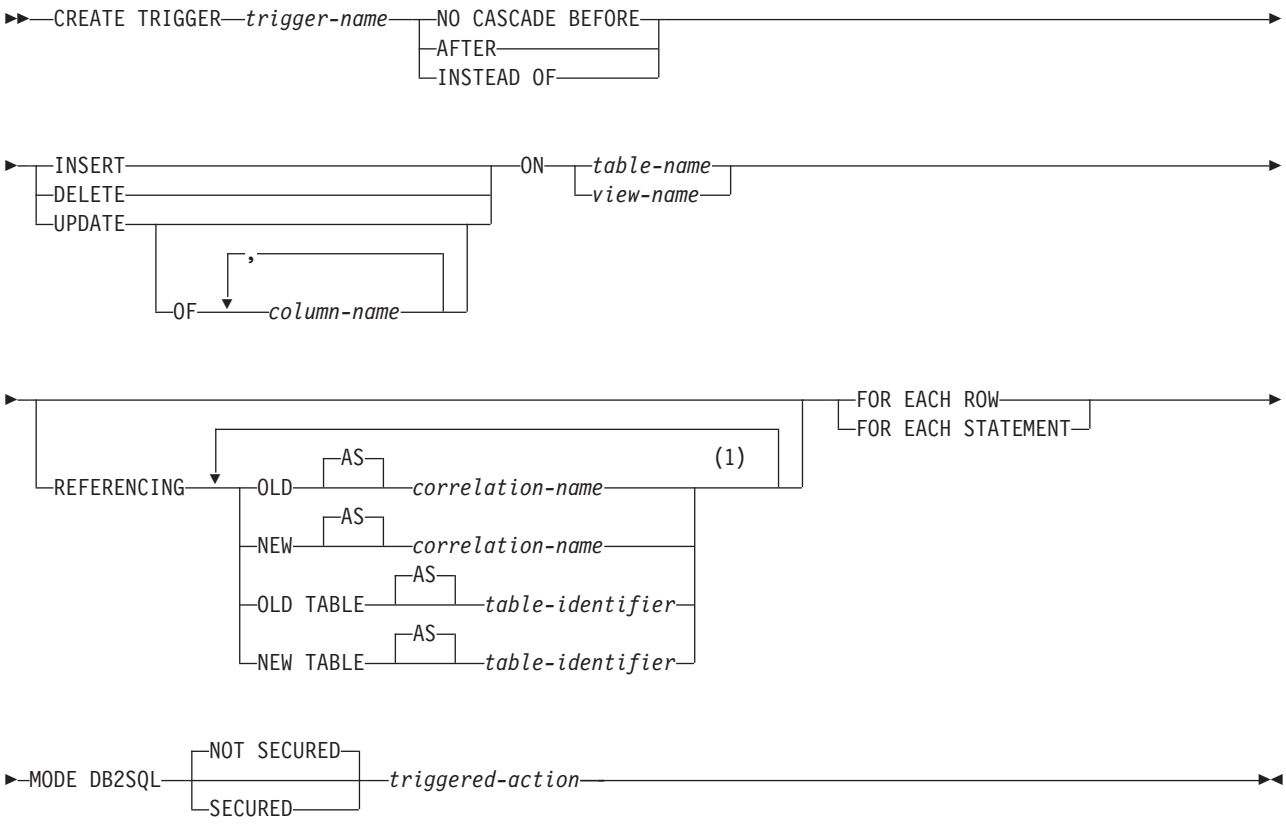
In defining a trigger on a table, the privileges held by the authorization ID of the statement must also include at least one of the following:

- Each of the following:
  - The ALTER privilege on the table on which the trigger is defined.
  - The UPDATE privilege on the table on which the trigger is defined, for DB2 for i, if the BEFORE UPDATE trigger contains a SET statement that modifies the NEW correlation variable.
- Database administrator authority.

In defining a trigger on a view, the privileges held by the authorization ID of the statement must also include at least one of the following:

- Ownership of the view.
- Database administrator authority.

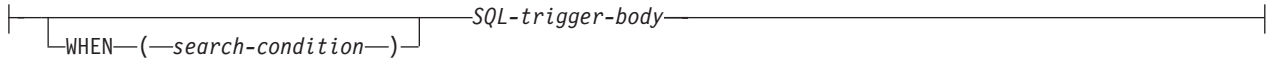
Syntax



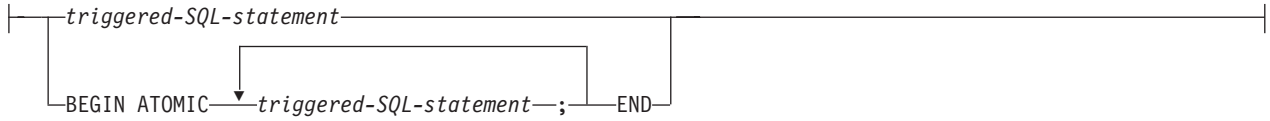
Notes:

1 The same clause must not be specified more than once.

triggered-action:



SQL-trigger-body:



Description

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. If a qualified trigger name is specified, the *schema-name* must not be one of the system schemas (see "Schemas" on page 11).

NO CASCADE BEFORE
Specifies that the trigger is a *before* trigger. The database manager executes the *triggered-action* before it applies any changes caused by its applicable 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.

NO CASCADE BEFORE must not be specified when *view-name* is also 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 its applicable 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 type of operation on a given subject view. The database manager executes the triggered-action instead of the insert, update, or delete operation on the subject view.

INSTEAD OF must not be specified when *table-name* is also 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.

INSERT
Specifies that the trigger is an *insert* trigger. The database manager executes the *triggered-action* whenever there is an insert operation on the subject table.

DELETE
Specifies that the trigger is a *delete* trigger. The database manager executes the *triggered-action* 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 trigger is an *update* trigger. The database manager executes the *triggered-action* 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.

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 must not be specified for an INSTEAD OF trigger.

ON *table-name*
Identifies the subject table of the 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, an alias or a declared temporary table.

ON *view-name*
Identifies the subject view of the INSTEAD OF trigger definition. The name must identify a view that exists at the current server, but must not identify a view where any of the following conditions are true:

- The view is defined with the WITH CASCADED CHECK option (a symmetric view).
- The view is a catalog view.
- The view has a column that is a LOB or XML (or a distinct type that is defined as a LOB).
- The view has a column that is based on an underlying column that is defined as an identity column or row change timestamp column.
- The view has a column that is based on a column of a result table that involves a set operator.
- The view has other views that are dependent on it.
- G DB2 for z/OS restricts the view from having a column that is defined G (directly or indirectly) as an expression.

REFERENCING

Specifies the correlation names for the transition variables 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. Transition variables with XML types cannot be referenced inside of a trigger. If the column of a transition table is referenced, the data type of the column cannot be XML.

Each row affected by the triggering SQL operation is available to the *triggered-action* by qualifying columns with *correlation-names* specified as follows:

OLD AS *correlation-name*
Specifies a correlation name that identifies the values in the row prior to the triggering SQL operation.

NEW 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.

The complete set of rows affected by the triggering SQL operation is available to the *triggered-action* by using *table-identifiers* specified as follows:

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.

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 valid 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 *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 *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 and NEW *correlation-name* variables cannot be modified in an AFTER trigger or INSTEAD OF trigger.

The table below summarizes the allowable combinations of transition variables and transition tables.

Granularity	Activation Time	Triggering Operation	Transition Variables Allowed	Transition Tables Allowed
FOR EACH ROW	BEFORE	DELETE	OLD	NONE
		INSERT	NEW	
		UPDATE	OLD, NEW	
	AFTER or INSTEAD OF	DELETE	OLD	OLD TABLE
		INSERT	NEW	NEW TABLE
		UPDATE	OLD, NEW	OLD TABLE, NEW TABLE
FOR EACH STATEMENT	BEFORE	DELETE	NONE	NONE
		INSERT		
		UPDATE		
	AFTER	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 and 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 or an INSTEAD OF trigger.

MODE DB2SQL
Specifies the mode of the trigger. MODE DB2SQL triggers are activated after all of the row operations have occurred.

NOT SECURED or SECURED
Specifies whether the trigger is considered secure for row access control and column access control.

NOT SECURED
Specifies that the trigger is considered not secure for row access control and column access control. This is the default.

NOT SECURED must not be specified explicitly or implicitly for a trigger whose subject table is using row access control or column access control. NOT SECURED must also not be specified for a trigger that is created for a view and row or column access control is active for one or more of the underlying tables in the view definition.

SECURED
Specifies that the function is considered secure for row access control and column access control.

SECURED must be specified for a trigger whose subject table is using row access control or column access control. SECURED must also be specified for a trigger that is created for a viewand row or column access control is active for one or more of the underlying tables in the view definition.

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.

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.

The *search-condition* for a before trigger must not include a subselect that references the subject table.

The WHEN clause must not be specified for INSTEAD OF triggers.

A reference to a transition variable with an XML data type cannot be used.

SQL-trigger-body
Specifies the SQL statements that are to be executed for the *triggered-action*.

triggered-SQL-statement
Specifies a single SQL statement that is to be executed for the *triggered-action*.

BEGIN ATOMIC *triggered-SQL-statement; ...* **END**
Specifies a list of SQL statements that are to be executed for the *triggered-action*. The statements are executed in the order in which they are specified.

Only certain SQL statements can be specified in the *SQL-trigger-body*. The following table shows the list of allowable SQL statements, which differs depending on whether the trigger is defined as BEFORE, AFTER, or INSTEAD OF. An 'X' in the table indicates that the statement is valid.

SQL statement	BEFORE	AFTER	INSTEAD OF
"CALL" on page 584	X [117]	X	X
"fullselect" on page 500	X	X	X
"REFRESH TABLE" on page 845		X	X
"SET transition-variable" on page 888	X		
"SIGNAL statement" on page 953	X	X	X
"VALUES" on page 902	X	X	X
"INSERT" on page 810		X	X
"MERGE" on page 820		X	X
Searched "DELETE" on page 764		X	X
Searched "UPDATE" on page 894		X	X

All tables, views, aliases, user-defined 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.

A fullselect specified in a before trigger must not refer to the subject table of the trigger.

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 *table-identifiers* that are specified in the *select-statement*, the name identifies the common table expression that is in the innermost scope.
- If the unqualified name corresponds to a transition table *table-identifier*, 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.

Notes

Owner privileges: There are no specific privileges on a trigger. When an INSTEAD OF trigger is defined, the associated privilege (INSERT, UPDATE, or DELETE on the view) is granted to the owner of the view. The owner is granted the privilege with the ability to grant that privilege to others. For more information on ownership of an object, see "Authorization, privileges and object ownership" on page 21.

Execution authorization: The user executing the triggering SQL operation does not need authority to execute a *triggered-SQL-statement*. A *triggered-SQL-statement* will execute using the authority of the *owner* of the trigger.

Activating a trigger: Only insert, delete, or update operations can activate a trigger. 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

117. The procedure must not have an attribute of MODIFIES SQL DATA.

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 operation. The number of levels of nested trigger cascading is limited to 16. For more information see Chapter 9, "SQL limits," on page 959.

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: 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),
  - 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),
  - 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.

Creating a trigger with the SECURED option: The trigger is considered secure after the CREATE TRIGGER statement is executed. DB2 treats the SECURED attribute as an assertion that declares that the user has established an audit procedure for all activities in the trigger body. If a secure trigger references user-defined functions, DB2 assumes those functions are secure without validation. If those functions can access sensitive data, a user that has security administrator authority needs to ensure that those functions are allowed to access that data and that an audit procedure is in place for those functions, and that all subsequent ALTER FUNCTION statements are being reviewed through this audit process.

A trigger must be secure if row or column access control is active for the triggering table. SECURED must also be specified for a trigger that is created for a view and row or column access control is active for one or more of the underlying tables in the view definition.

Creating a trigger with the NOT SECURED option: The CREATE TRIGGER statement returns an error if row or column access control is active for the triggering table, or if its triggering table is a view and row or column access control is active for one or more of the underlying tables in the view definition.

Transition variable values and row and column access control: Row and column access control is not enforced for transition variables and transition tables. If row or column access control is enforced for the triggering table, row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row and column access control enforced for the triggering table is also ignored for transition variables and transition tables that are referenced in the trigger body or are passed as arguments to user-defined functions invoked within the trigger body. To ensure there are no security concerns for SQL statements accessing sensitive data in transition variables and transition tables, the trigger must be created with the SECURED option. If the trigger is not secure, row access control and column access control cannot be enforced for the triggering table and the CREATE TRIGGER statement returns an error.

Multiple triggers: Multiple triggers that have the same triggering SQL operation and activation time can be defined on a table. The triggers are activated in the order in which they were created. For example, the trigger that was created first is executed first, the trigger that was created second is executed second.

A maximum of 300 triggers can be added to any given table. For more information see Chapter 9, “SQL limits,” on page 959.

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 subject table is referenced in the *triggered-action*, 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 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 that include those references to the object will be rebuilt when the trigger is activated. 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 activated. If the appropriate privilege does not exist at that time, the corresponding insert, update or delete operation on the subject table will fail.

G DB2 for LUW effectively drops the trigger when a dependent object is dropped or
G a required privilege is revoked.

Errors executing triggers: If a SIGNAL statement is executed in the *SQL-trigger-body*, the SQLSTATE specified in the SIGNAL statement will be returned.

Other errors that occur during the execution of *SQL-trigger-body* statements are typically returned using SQLSTATE 09000.

Special registers: 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
G registers are inherited from the triggering SQL operation. In DB2 for z/OS, the
G values of special registers are only inherited from the triggering SQL operation if
G they have been explicitly set in the connection.

Transaction isolation: All the statements in the *SQL-trigger-body* run under the
G isolation level of the triggering SQL operation. In DB2 for z/OS the SQL
G statements in the *SQL-trigger-body* run under the isolation level used at the time the
G trigger was created.

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

The CREATE TYPE statement defines a user-defined data type at the current server. The following types of user-defined data types can be defined:

- Array

  A user-defined data type that is an ordinary array. The elements of an array type are based on one of the built-in data types. See "CREATE TYPE (array)" on page 730.

- Distinct

  A user-defined data type that shares a common representation with one of the built-in data types. Functions that cast between the user-defined distinct type and the source built-in data type are generated when the user-defined distinct type is created. Optionally, support for comparison operations to use with the user-defined distinct type can be generated when the user-defined distinct type is created. See "CREATE TYPE (distinct)" on page 734.

CREATE TYPE (array)

The CREATE TYPE (array) statement defines an array type at the current server.

Invocation

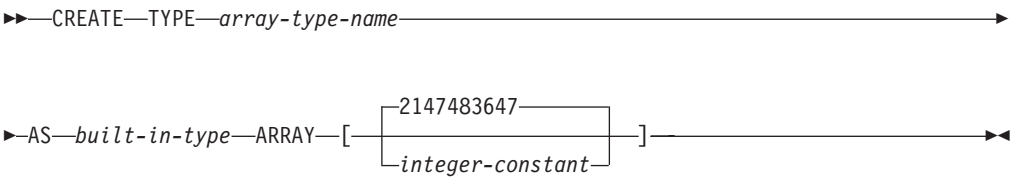
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

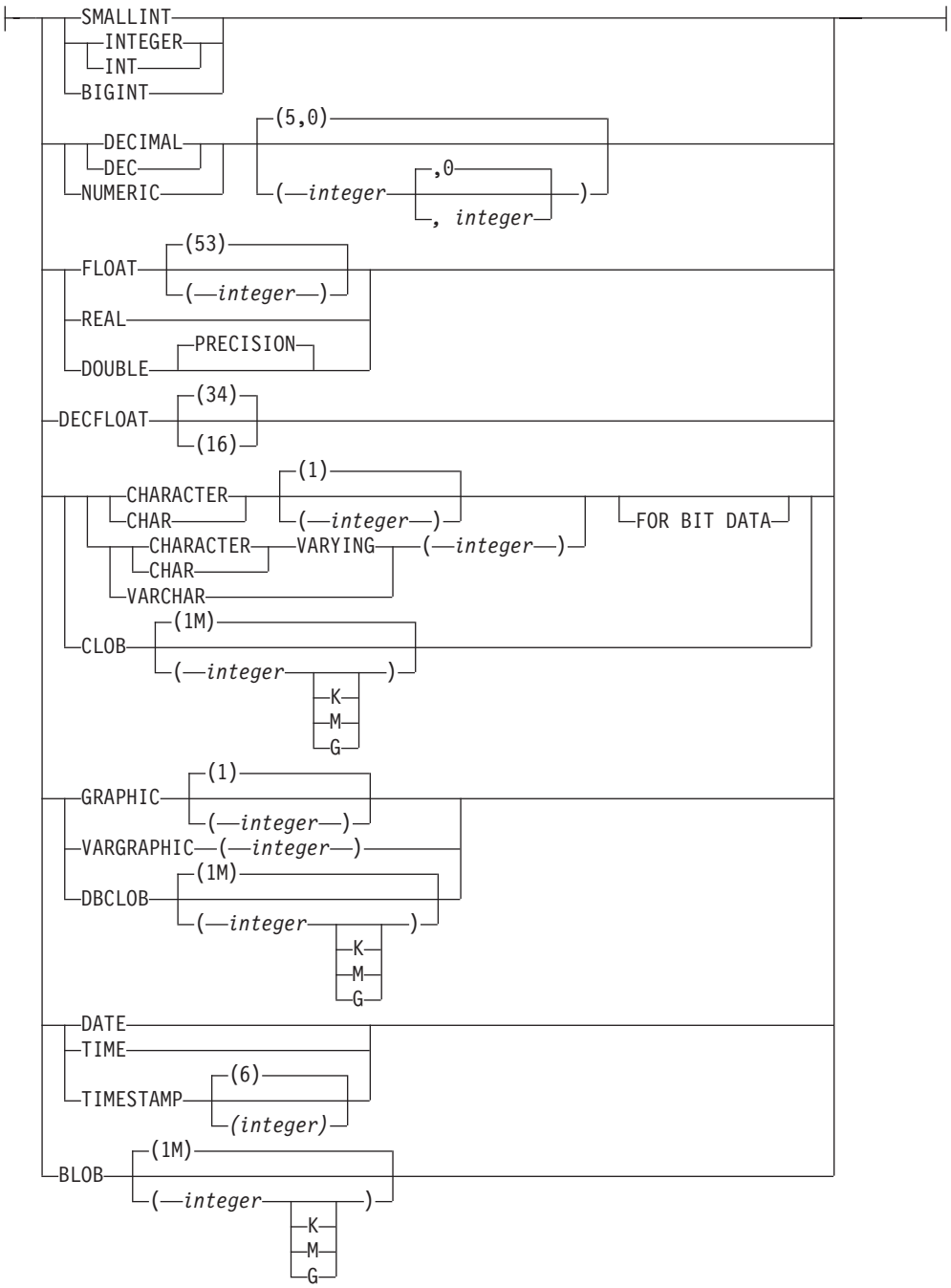
The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema
- Database administrator authority.

Syntax



built-in-type:



Description

array-type-name
Names the array type. The name, including the implicit or explicit qualifier must not identify a distinct type or array type that already exists at the current server. *distinct-type-name* must not be the same as the name of a built-in data type, or any of the following, even if they are specified as delimited identifiers:

ALL	HASHED_VALUE	SOME	XMLNAMESPACES
AND	IN	STRIP	XMLPARSE
ANY	INTERVAL	SUBSTRING	XMLPI
ARRAY_AGG	IS	TABLE	XMLROW
BETWEEN	LIKE	THEN	XMLSERIALIZE
BINARY	MATCH	TRIM	XMLTEXT

BOOLEAN	NODENAME	TRUE	XMLVALIDATE
CASE	NODENUMBER	TYPE	XSLTRANSFORM
CAST	NOT	UNIQUE	=
CHECK	NULL	UNKNOWN	¬=
DATAPARTITIONNAME	ONLY	VARBINARY	<=
DATAPARTITIONNUM	OR	WHEN	<
DBPARTITIONNAME	OVERLAPS	XMLAGG	>
DBPARTITIONNUM	PARTITION	XMLATTRIBUTES	¬<
DISTINCT	POSITION	XMLCOMMENT	¬>
EXCEPT	RID	XMLCONCAT	>=
EXISTS	ROWID	XMLDOCUMENT	!<
EXTRACT	RRN	XMLELEMENT	<>
FALSE	SELECT	XMLFOREST	!>
FOR	SIMILAR	XMLGROUP	!=
FROM			

If a qualified *array-type-name* is specified, the schema name must not be one of the system schemas (see "Schemas" on page 11).

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 688 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 array type is sourced on a string data type, a CCSID is associated with the array type at the time the array type is created.

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. Each varying length string array element is allocated as its maximum length. The maximum number of array elements that can be assigned in the array is the number of elements that fit in 4 gigabytes.

Notes

Owner privileges: The owner of the array type is authorized to define parameters or variables with the array type with the ability to grant these privileges to others. See "GRANT (type privileges)" on page 804. For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

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

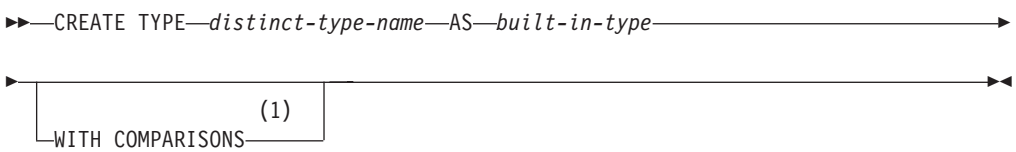
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

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
- Database administrator authority.

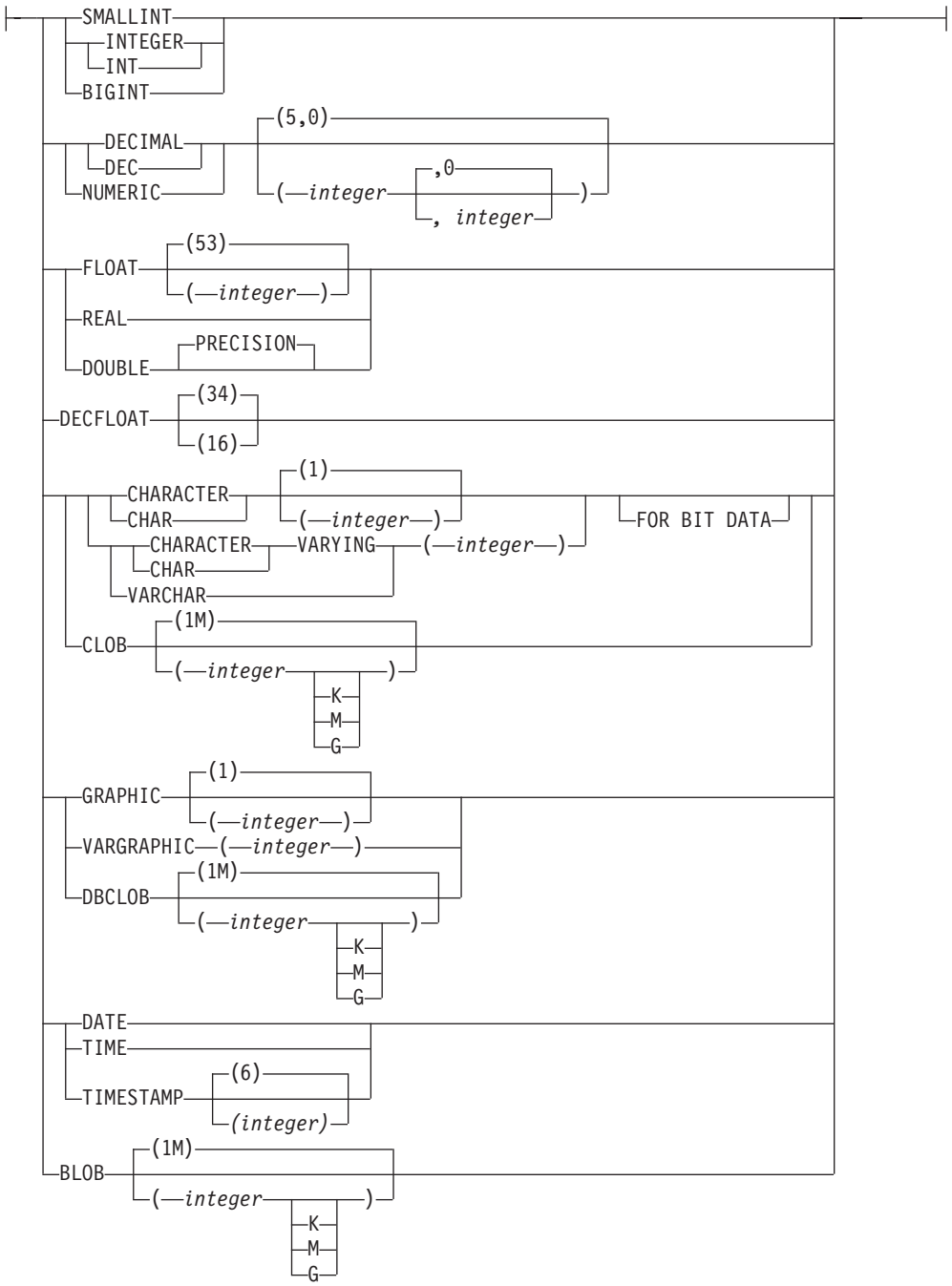
Syntax



Notes:

1 Specify WITH COMPARISONS for built-in data types except for BLOB, CLOB, and DBCLOB.

built-in-type:



Description

distinct-type-name

Names the distinct type. The name, including the implicit or explicit qualifier must not identify a distinct type or array type that already exists at the current server. *distinct-type-name* must not be the same as the name of a built-in data type, or any of the following, even if they are specified as delimited identifiers:

ALL	HASHED_VALUE	SOME	XMLNAMESPACES
AND	IN	STRIP	XMLPARSE
ANY	INTERVAL	SUBSTRING	XMLPI
ARRAY_AGG	IS	TABLE	XMLROW
BETWEEN	LIKE	THEN	XMLSERIALIZE
BINARY	MATCH	TRIM	XMLTEXT

```
BOOLEAN             NODENAME          TRUE              XMLVALIDATE
CASE                NODENUMBER        TYPE              XSLTRANSFORM
CAST                NOT               UNIQUE            =
CHECK               NULL              UNKNOWN           ¬=
DATAPARTITIONNAME   ONLY              VARBINARY         <=
DATAPARTITIONNUM    OR                WHEN              <
DBPARTITIONNAME     OVERLAPS          XMLAGG            >
DBPARTITIONNUM      PARTITION         XMLATTRIBUTES     ¬<
DISTINCT            POSITION          XMLCOMMENT        ¬>
EXCEPT              RID               XMLCONCAT         >=
EXISTS              ROWID             XMLDOCUMENT       !<
EXTRACT             RRN               XMLELEMENT        <>
FALSE               SELECT            XMLFOREST         !>
FOR                 SIMILAR           XMLGROUP          !=
FROM
```

If a qualified *distinct-type-name* is specified, the schema name must not be one of the system schemas (see "Schemas" on page 11).

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 688 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.

WITH COMPARISONS
Specifies that system-generated comparison operators are to be created for comparing two instances of a distinct type. Do not specify these keywords if the built-in type is BLOB, CLOB, or DBCLOB, otherwise a warning will be returned and the comparison operators will not be generated. For all other *built-in-type*s, the WITH COMPARISONS keywords are required.

When a distinct type is created using the WITH COMPARISONS clause, the database manager allows the comparison operators with the exception of LIKE and NOT LIKE. In order to use the LIKE predicate on a distinct type, it must be cast to a built-in type. The comparison operators are invoked as infix operators, not by using functional notation; that is, C1 < C2, not "<"(C1,C2).

Notes

Owner privileges: The owner of the distinct type is authorized to define columns, parameters, or variables with the distinct type with the ability to grant these privileges to others. See "GRANT (type privileges)" on page 804. The owner is also authorized to invoke the generated cast functions (see "GRANT (function or procedure privileges)" on page 793). For more information on ownership of the object, see "Authorization, privileges and object ownership" on page 21.

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 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:

```
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 53 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 type in the CREATE TYPE statement, 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. 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 including the schema qualfier. 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 as the generated cast function must not already exist in the current server.

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.

The following table gives the names of the functions to convert from the distinct type to the source type and from the source type to the distinct type for all predefined data types.

Table 53. 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

Table 53. CAST functions on distinct types (continued)

Source Type Name	Function Name	Parameter-type	Return-type
DECIMAL	*distinct-type-name*	DECIMAL (*p,s*)	*distinct-type-name*
	DECIMAL	*distinct-type-name*	DECIMAL (*p,s*)
G NUMERIC or DECIMAL[1] G	*distinct-type-name*	NUMERIC (*p,s*) or DECIMAL (*p,s*)	*distinct-type-name*
G G	NUMERIC or DECIMAL	*distinct-type-name*	NUMERIC (*p,s*) or DECIMAL (*p,s*)
REAL or FLOAT(*n*) where *n* defines a single precision floating point number	*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 or FLOAT(*n*) where *n* defines a double precision floating point number	*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*
	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*)
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

Note:

G 1. When the source data type is specified as NUMERIC, whether a separate function named NUMERIC is generated
G is platform-specific. DB2 for i generates a cast function named NUMERIC. DB2 for z/OS and DB2 for LUW do
G not generate a cast function named NUMERIC, use DECIMAL 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.

The schema name of the distinct type must be included in the SQL path for successful use of these operators and cast functions in SQL statements.

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

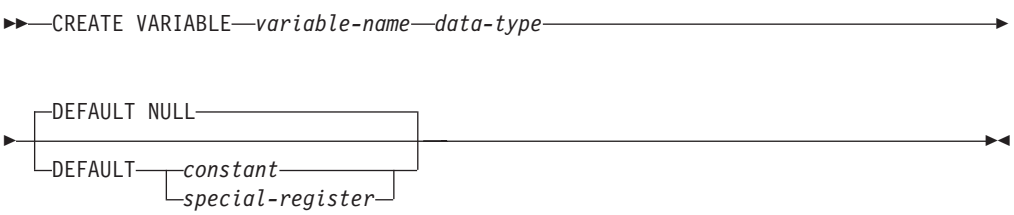
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

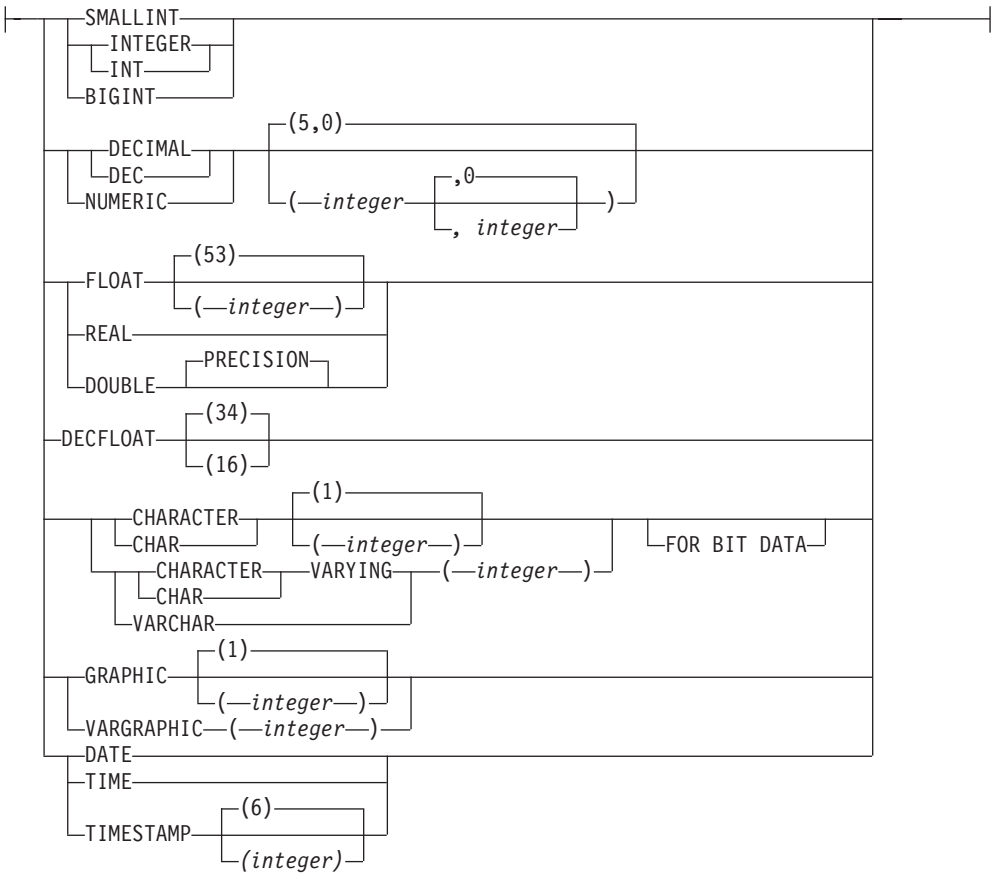
- The privilege to create in the schema
- Database administrator authority.

Syntax



data-type:

built-in-type:



Description

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. The schema name must not be a system schema.

data-type
Specifies the data type or the global variable.

built-in-type
Specifies the built-in data type used as the basis for the internal representation of the global variable. See "CREATE TABLE" on page 688 for a more complete description of each built-in data type.

DEFAULT
Specifies a default value for the global variable. The value can be a constant, a special register, or the keyword NULL. The default value is determined on its first reference if a value is not explicitly specified. If a default value is not specified, the variable is initialized to the null value. The default value must be assignment-compatible with the data type of the variable.

Notes

Owner privileges: The owner always acquires the READ and WRITE privilege on the variable with the ability to grant these privileges to others.

Session scope: 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: 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.

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.

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.

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.

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.

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'
```

CREATE VIEW

The CREATE VIEW statement defines 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

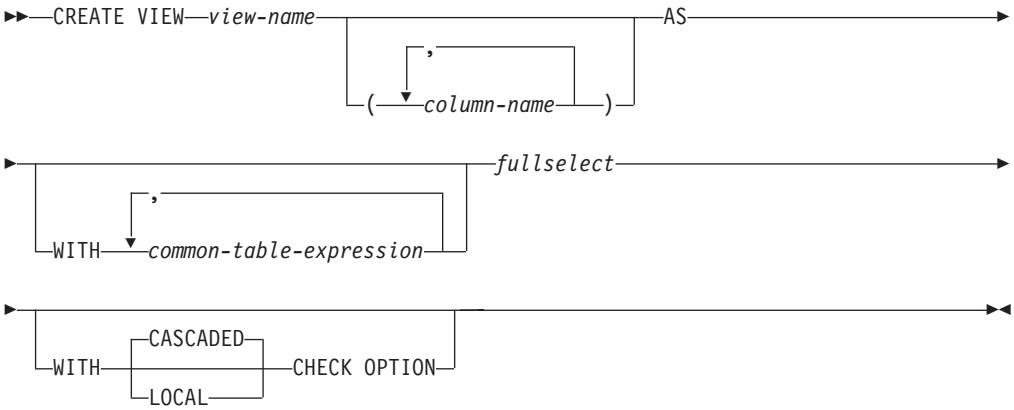
The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema
- Database administrator authority.

The privilege held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the *fullselect*:
  - The SELECT privilege on the table or view
  - Ownership of the table or view.
- Database administrator authority.

Syntax



Description

view-name
Names the view. The name, including the implicit or explicit qualifier, must not identify an alias, index, table or view that already exists at the current server.

(*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* 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 of the result table of the *fullselect*.

A list of column names must be specified if the result table of the *fullselect* has duplicate column names or an unnamed column. For more information about unnamed columns, see "Names of result columns" on page 468.

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 506.

fullselect
At any time, the view consists of the rows that would result if the *fullselect* were executed.

The *fullselect* must not reference a host variable, SQL variable, or SQL parameter, but may reference global variables.

A temporary table must not be referenced in the *fullselect*

The *fullselect* must not contain a *data-change-table-reference*.

In DB2 for z/OS, a view that has an INSTEAD OF trigger must not be referenced in the *fullselect*.

The maximum number of columns allowed in a view is 750. The maximum number of base tables allowed in a view is 225. See Table 63 on page 967 for more information.

For an explanation of *fullselect*, see "fullselect" on page 500.

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.

The CHECK OPTION clause must not be specified:

- if the view is read-only or includes a subquery.
- if the view references a non-deterministic function.
- if the view references another view and that view has an INSTEAD OF trigger.
- if the definition of the view contains a special register in other than the outer select list of the view
- if the view is recursive

If the CHECK OPTION clause is specified for an updatable view that does not allow inserts, then it applies to updates only.

If the CHECK OPTION clause 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 the CHECK OPTION clause. Because the definition of the view is not used, rows might be inserted or updated through the view that do not conform to the definition of 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 V is an underlying view for an updatable view, the CHECK OPTION clause on V also applies to that view, even if the CHECK OPTION clause is not specified on that view. The search conditions of V and each view which is an underlying view for V are ANDed together to form a search condition that is applied for an insert or update of V or of any view dependent on V.

Consider the following updatable views which shows the impact of the WITH CASCADED CHECK OPTION:

```
CREATE VIEW V1 AS SELECT COL1 FROM T1 WHERE COL1 > 10

CREATE VIEW V2 AS SELECT COL1 FROM V1 WITH CASCADED 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

The WITH LOCAL CHECK OPTION on a view V means the search condition of V is applied as a constraint for an insert or update of V or of any view that is dependent on V. 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 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
```

This example shows V1 as *an underlying view for* V2 and V2 *dependent on* V1.

The following table describes which search conditions are checked during an INSERT or UPDATE operation:

Table 54. Views whose search conditions are checked during INSERT and UPDATE

View used in INSERT or UPDATE operation	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

Notes

Owner privileges: The owner always acquires the SELECT privilege on the view. If all of the privileges that are required to create the view are held with the GRANT option before the view is created, the owner of the view receives the SELECT privilege with the GRANT option. Otherwise, the owner receives the SELECT privilege without the GRANT option. For example, assume that a view is created on a table for which the owner has the SELECT privilege with the GRANT option and the view definition also refers to a user-defined function. If the owner's EXECUTE privilege on the user-defined function is held without the GRANT option, the owner acquires the SELECT privilege on the view without the GRANT option.

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*. The privileges can be granted only if the privileges from which they are derived also can be granted. The owner only acquires these privileges if the privileges from which they are derived exist at the time the view is created. For more information on ownership of objects see “Authorization, privileges and object ownership” on page 21.

Deletable views: A view is *deletable* if an INSTEAD OF DELETE trigger has been defined for the view, or if all of the following are true:

- The FROM clause of the outer *fullselect* identifies only one base table, deletable view, or deletable nested table expression (that is, a nested table expression whose *fullselect*, if used to create a view, would create a deletable view) that is not a catalog table or view.
- 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 DISTINCT.
- No base table (or underlying base table of a view) in a subquery contained in the *fullselect* is the same as the base table (or underlying base table of a view) in the outer *fullselect*.

Updatable views: A view is *updatable* if an INSTEAD OF UPDATE trigger 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 UPDATE trigger has been defined for the view, or if the corresponding result column of the *fullselect* 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 INSERT trigger has been defined for the view, or if, independent of an INSTEAD OF trigger for update, at least one column of the view is updatable.

If a view contains two updatable columns that refer to the same column in the underlying table, without an INSTEAD OF trigger for insert the view is not insertable.

Read-only views: A view is *read-only* if it is NOT deletable, updatable, or insertable.

A read-only view cannot be the object of an INSERT, UPDATE, or DELETE statement.

Unqualified table names: 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 *table-identifiers* that are specified in the *select-statement*, 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.

Examples

Example 1: Create a view named MA_PROJ upon 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 only one of the column names is being changed, 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 RESPEMP 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, RESPEMP 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
```

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

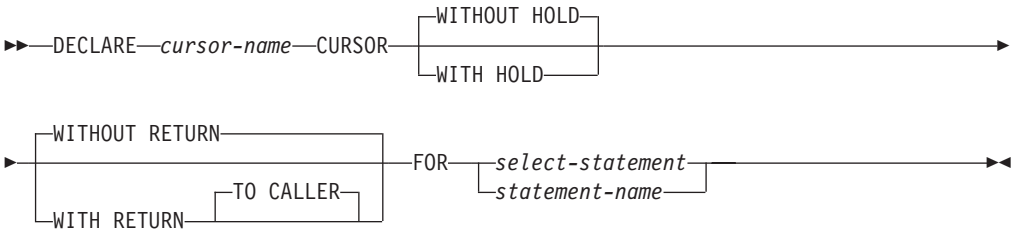
No authorization is required to use this statement. However if OPEN or FETCH is used for the cursor, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the SELECT statement of the cursor:
  - The SELECT privilege on the table or view
  - Ownership of the table or view
- Database administrator authority.

The SELECT statement of the cursor can have one of the following forms:

- The prepared *select-statement* identified by the *statement-name*. In this case:
  - The authorization ID of the statement is the run-time authorization ID.
  - The authorization check is performed when the *select-statement* is prepared.
  - The cursor cannot be opened unless the *select-statement* is successfully prepared.
- The specified *select-statement*. In this case:
  - The authorization ID of the statement is the authorization ID specified during program preparation.
  - In REXX, the authorization ID of the statement is the run-time authorization ID.
  - Depending on the product environment or options, the authorization check is performed either during program preparation, or when the cursor is opened. See the product references for further information.

Syntax



Description

cursor-name

Names the cursor. The name must not be the same as the name of another cursor declared in the source program.

WITHOUT HOLD or WITH HOLD
Specifies whether the cursor should be prevented from being closed as a consequence of a commit operation.

WITHOUT HOLD
Does not prevent the cursor 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, this is the default.

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 by a commit only if the connection 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. Afterwards, 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. 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.

WITHOUT RETURN
Specifies that the result table of the cursor is not 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, this is the default.

WITH RETURN TO CALLER
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.

WITH RETURN TO CALLER is relevant when the SQL CALL statement is used to invoke a procedure that either contains the DECLARE CURSOR statement, or directly or indirectly invokes a program or procedure that contains the DECLARE CURSOR statement. In other cases, the precompiler might accept the clause, but the clause has no effect.

When a cursor that is declared using the WITH RETURN TO CALLER clause remains open at the end of a program or procedure, that cursor defines a result set from the program or procedure. Use the CLOSE statement to close cursors that are not intended to be a result set from the program or procedure.

The result set consists of all rows from the current cursor position to the end of the result table.

For Java external procedures, all cursors are implicitly declared WITH RETURN TO CALLER.

select-statement
Specifies the SELECT statement of the cursor. See "select-statement" on page 505 for more information.

The *select-statement* must not include parameter markers (except for REXX), but can include references to other variables. In host languages, other than REXX, the declarations of the variables must precede the DECLARE CURSOR statement in the source program. In REXX, parameter markers must be used in place of host variables and the statement must be prepared.

statement-name
Specifies the prepared *select-statement* that specifies the result table of the cursor whenever 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 833 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.

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: [118]

- The FROM clause of the outer *fullselect* identifies only one base table or deletable view (cannot identify a nested table expression).
- 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.
- The *select-clause* of the outer *fullselect* does not include DISTINCT.
- The *select-statement* does not include an ORDER BY clause.
- The *select-statement* does not include a READ ONLY clause.
- The FROM clause of the outer fullselect does not include a *data-change-table-reference*.
- The *select-statement* does not include a FETCH FIRST n ROWS ONLY clause.
- The result of the outer *fullselect* does not make use of a temporary table.
- No base table (or underlying base table of a view) in a subquery contained in the *fullselect* is the same as the base table (or underlying base table of a view) in the outer *fullselect*.
- If it is executed with isolation level UR, then the UPDATE clause must be specified.

G In DB2 for z/OS, a cursor is not *deletable* if the FROM clause of the outer *fullselect*
G identifies a view with instead of triggers.

118. In DB2 for z/OS and DB2 for LUW, a program preparation option must be used if the UPDATE clause is not specified in the *select-statement* of the cursor and for DB2 for LUW if the cursor is not statically defined. For DB2 for z/OS, use the precompiler option STDSQL(YES) or NOFOR. For DB2 for LUW, use the program preparation option LANGLEVEL SQL92E.

A column in the select list of the outer *fullselect* associated with a cursor is *updatable* if all of the following are true:[118]

- 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 UPDATE is specified without a list of column names, only the updatable columns in the SELECT clause of the *subselect* can be updated. If the UPDATE clause of the *select-statement* of the cursor is specified with a list of column names, only the columns specified in the list of column names can be updated.

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 in 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. Cursors that specify WITH RETURN TO CALLER in a procedure and are left open are returned as result sets.

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 to be returned 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 may block data for read-only cursors. If a cursor is not going to be used in a Positioned UPDATE or Positioned 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.

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;
    .
    .
    .
  }
```

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 and instance of the temporary table. When the application process terminates, 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 clause or AS (fullselect) 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 in the fullselect:

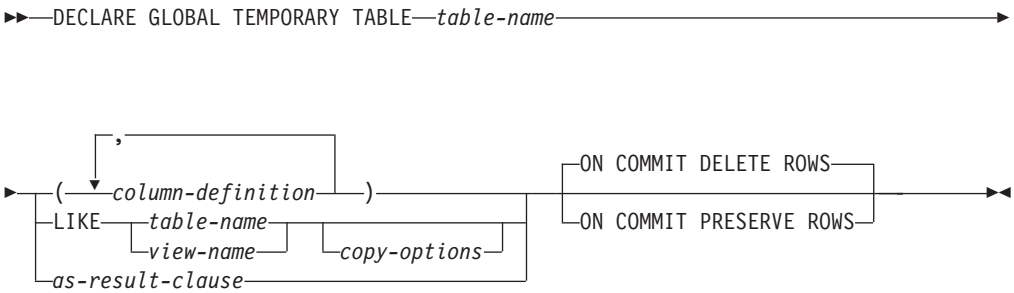
- The SELECT privilege for the table or view
- Ownership of the table or view
- Database administrator authority.

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

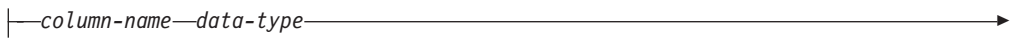
- The USAGE privilege on the distinct type
- Ownership of the distinct type
- Database administrator authority.

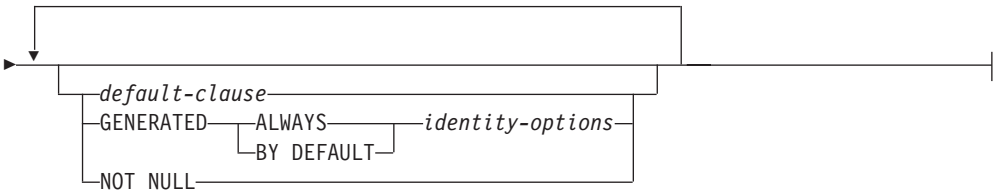
In DB2 for LUW, the user must have the privilege to create in a temporary table space to use this statement.

Syntax

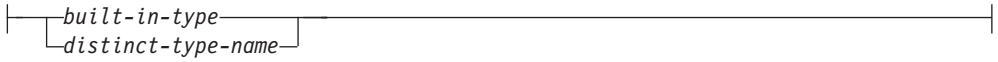


column-definition:

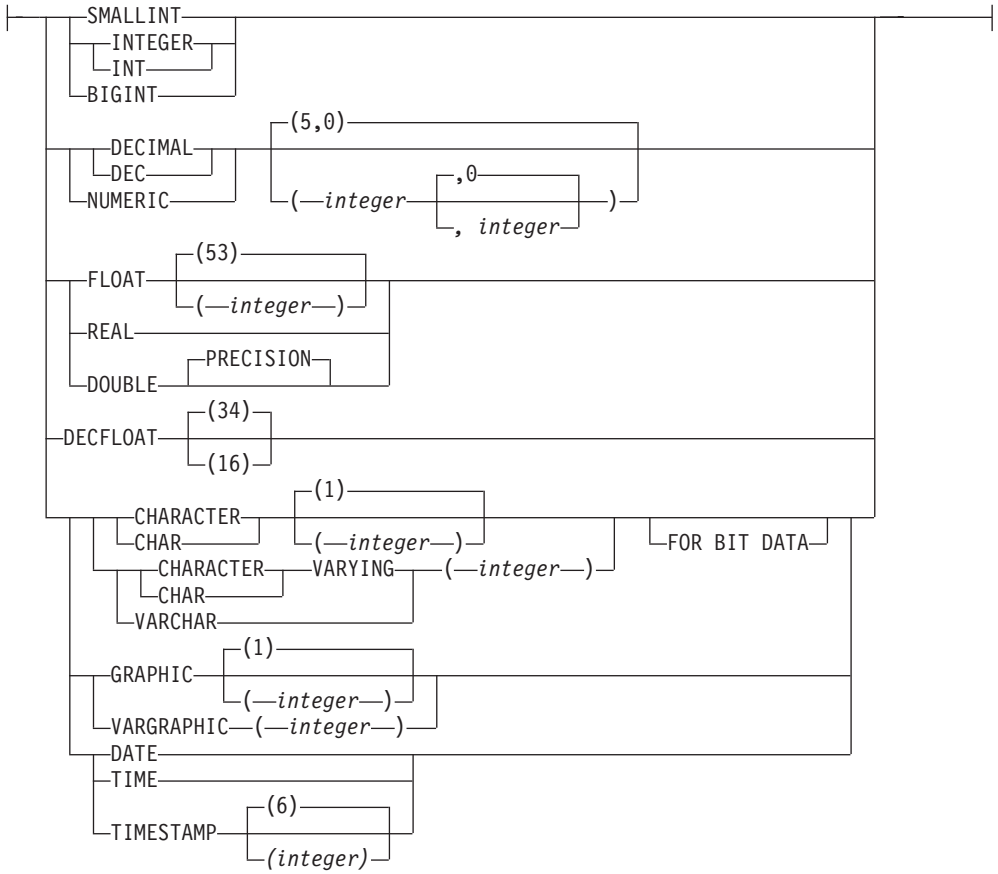




data-type:

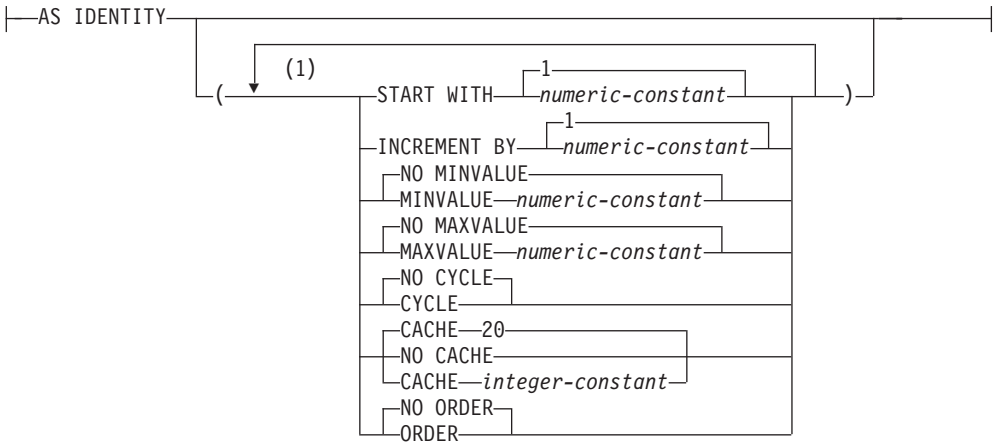


built-in-type:



default-clause:

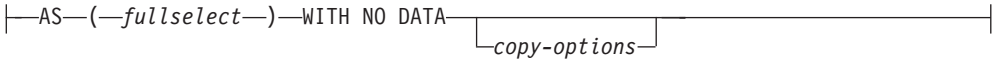
identity-options:



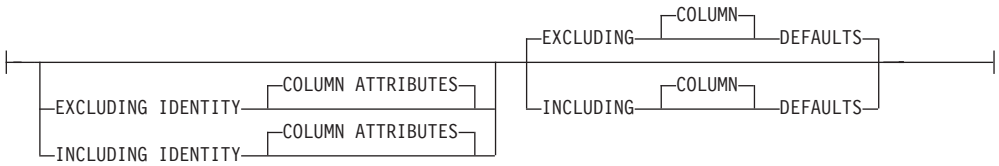
Notes:

1 The same clause must not be specified more than once.

as-result-clause:



copy-options:



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, the database manager implicitly defines it to be SESSION. If a declared temporary table, or an index 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 as SESSION.*table-name*.
- 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*.

column-definition

Defines the attributes of a column. There must be at least one column definition and no more than 750 columns for the table. See Table 63 on page 967 for more information.

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.

data-type
Specifies the data type of the column.

built-in-type
Specifies a built-in data type. Note that CLOB, DBCLOB, BLOB, and XML must not be specified. See "CREATE TABLE" on page 688 for the description of built-in types.

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 734.

DEFAULT
Specifies a default value for the column. This clause must not be specified more than once in the same *column-definition*.

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 77. A 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 an SQL data change statement as the default 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.

NULL
Specifies null as the default for the column. If NOT NULL was specified, DEFAULT NULL must not be specified within the same *column-definition*.

If the value specified is not valid, an error is returned.

GENERATED
Specifies that the database manager generates values for the column. GENERATED must be specified if the column is to be considered an IDENTITY 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.

Defining a column as generated does not necessarily guarantee uniqueness of the values. To ensure uniqueness of the values, define a unique, single-column index on the generated column.

AS IDENTITY
Specifies that the column is the identity column for the table. A table can only have a single 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.

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 688 for the descriptions of the identity attributes.

NOT NULL
Prevents the column from containing null values. Omission of NOT NULL implies that the column can contain null values.

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. For DB2 for z/OS, *table-name* must not identify a declared temporary table.

The table must not contain a column having a LOB data type, an XML data type or a distinct type.

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 each of the columns of *table-name*, or result columns of *view-name* (if applicable to the data type).

- Column name
- Data type, length, precision and scale
- CCSID
- Nullability

For base tables, the default value, identity, row change timestamp, and hidden attribute are also included in the table definition. For a view, if the column of the underlying base table has a default value, then the effect is product-specific.

The implicit definition does not include any other 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 (*fullselect***)**
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 *fullselect* if the *fullselect* were to be executed. The use of AS (*fullselect*) is an implicit definition of *n* columns for the declared temporary table, where *n* is the number of columns that would result from the *fullselect*.

The implicit definition includes the following attributes of the *n* columns (if applicable to the data type):

- Column name
- Data type, length, precision and scale
- CCSID
- Nullability

The following attributes are not included (the default value and identity 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 *fullselect*.

The *fullselect* must not result in a column having a LOB data type, an XML data type or a distinct type.

The *fullselect* must not refer to variables or include parameter markers.

The *fullselect* must not contain a PREVIOUS VALUE or a NEXT VALUE expression.

WITH NO DATA
Specifies that the query is used only to define the attributes of the new table. The table is not populated using the results of the query and the REFRESH TABLE statement cannot be used.

The columns of the table are defined based on the definitions of the columns that result from the *fullselect*.

The *fullselect* must not:

- Result in a column having a BLOB, CLOB, or DBCLOB data type or a distinct type based on these data types.
- Contain PREVIOUS VALUE or a NEXT VALUE expression.

copy-options

These options specify whether or not to copy additional attributes of the result table definition (table, view, or fullselect).

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 attributes, if any, of the columns resulting from the fullselect, table-name or view-name. In general, the identity attributes are copied if the element of the corresponding column in the fullselect, table or view 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 that is an identity column. If the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified with the AS *fullselect* clause, the columns of the new table do not inherit the identity attribute in the following cases:

- The select list of the fullselect includes multiple instances of the name of an identity column (that is, selecting the same column more than once).
- The select list of the fullselect includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The fullselect includes a set operation (union).

If INCLUDING IDENTITY is not specified, the table will not have an identity column.

If the LIKE clause identifies a view, INCLUDING IDENTITY COLUMN ATTRIBUTES must not be specified.

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
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 column defaults for each updatable column of the definition of the source table are inherited. Columns that are not updatable do not have a default defined in the corresponding column of the created table.

If INCLUDING COLUMN DEFAULTS is not specified, whether or not default values are included depends on whether the LIKE clause or the AS (*fullselect*) clause was specified. For more information, see the description of the LIKE clause or the AS (*fullselect*) clause above.

G DB2 for z/OS does not support EXCLUDING COLUMN DEFAULTS and
G INCLUDING COLUMN DEFAULTS for the LIKE clause.

ON COMMIT
Specifies the action to be taken on the declared temporary table when a COMMIT operation is performed. The default is DELETE ROWS.

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.

Notes

Instantiation, scope and termination: For the explanations below, P denotes a application process and T is a declared temporary table in the application process P:

- An empty instance of T is created as a result of a DECLARE GLOBAL TEMPORARY TABLE statement executed in P.
- Any SQL statement in P can make reference to T; and any reference to T in P is a reference to that same instance of T.

  If a DECLARE GLOBAL TEMPORARY TABLE statement is specified within a compound statement, the scope of the declared temporary table is the application process, not just the compound statement. The table is not implicitly dropped at the end of the compound statement. A declared temporary table

cannot be defined multiple times by the same name in other compound statements in that application process, unless the table has been explicitly dropped.

- 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.
- Assuming that the ON COMMIT DELETE ROWS clause was specified implicitly or explicitly, then when a commit operation terminates a unit of work in P, and there is no open WITH HOLD cursor in P that is dependent on T, then the commit deletes all rows from T.
- When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes a modification to SESSION.T, then the changes to T are undone.

  When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes the declaration of SESSION.T, then the rollback drops the table T.

  If a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes the drop of a declared temporary table SESSION.T, then the rollback will undo the drop of the table.
- When the application process that declared T terminates, T is dropped.

Privileges: When a declared temporary table is defined, PUBLIC is implicitly granted all table privileges on the table and authority to drop the table. These privileges cannot be revoked. This enables any SQL statement in the application process to reference a declared temporary table that has already been defined in that application process.

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, COMMENT, GRANT, LOCK TABLE, RENAME or REVOKE statement.
- Be referenced in a CREATE ALIAS, CREATE FUNCTION (SQL Scalar), CREATE TRIGGER, or CREATE VIEW statement.
- Be specified in referential constraints.
- Be referenced in a CREATE INDEX statement unless the schema name of the index is SESSION.

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.

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 can 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:

- The DELETE privilege for the table or view
- Ownership of the table [119]
- Database administrator authority.

If *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 for the table or view [120]
- Ownership of the table or view
- Database administrator 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:

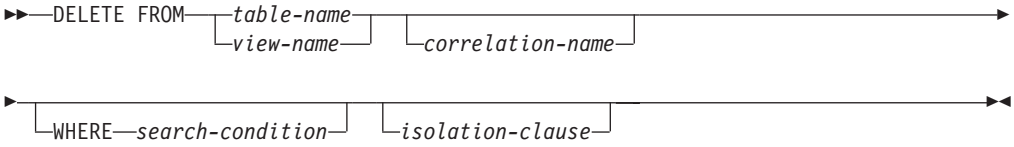
- For every table or view identified in the subquery:
  - The SELECT privilege on the table or view, or
  - Ownership of the table or view.
- Database administrator authority.

119. The DELETE privilege on a view is only inherent in database administrator authority. Ownership of a view does not necessarily include the DELETE privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.

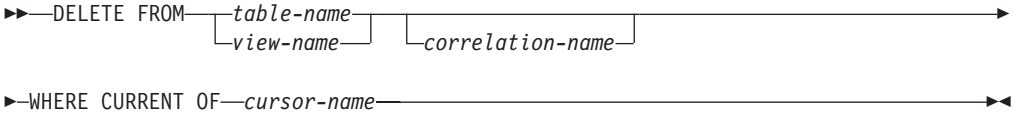
120. In DB2 for z/OS, and DB2 for LUW, the authorization ID of the statement only requires the DELETE privilege for the table or view. To require the SELECT privilege, a standards option must be in effect. For DB2 for z/OS use the precompiler option SQLRULES(STD) or set the CURRENT RULES special register to 'STD'. For DB2 for LUW, use the program preparation option LANGLEVEL SQL92E.

Syntax

Searched DELETE:



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 743.

correlation-name
Specifies an alternate name that can be used within the *search-condition* to designate the table or view. For an explanation of *correlation-name*, see "Correlation names" on page 108.

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
Specifies a search condition, as described in "Search conditions" on page 178. 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 *search-condition* is true.

If *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.

Let T2 denote the object table of a DELETE statement and let T1 denote a table that is referenced in the FROM clause of a subquery of that statement. T1 must not be a table that can be affected by the DELETE on T2. Thus, the following rules apply:

- T1 must not be a dependent of T2 in a relationship with a delete rule of CASCADE or SET NULL, unless the result of the subquery is materialized before the DELETE action is executed.
- T1 must not be a dependent of T3 in a relationship with a delete rule of CASCADE or SET NULL if deletes of T2 cascade to T3.

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* section of "DECLARE CURSOR" on page 749.

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 749.

When the DELETE statement is executed, the cursor must be open and positioned on a row and that row is 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.

G In DB2 for z/OS, if the DELETE statement is embedded in a program, the
G DECLARE CURSOR statement must include a *select-statement* rather than a
G *statement-name*.

G In DB2 for z/OS, a positioned DELETE must not be specified for a cursor
G that references a view on which an instead of delete trigger is defined,
G even if the view is a deletable view.

isolation-clause

Specifies the isolation level used by the statement.

WITH

Introduces the isolation level, which may be one of:
- *RR* Repeatable read
- *RS* Read stability
- *CS* Cursor stability

If *isolation-clause* is not specified, the default isolation is used. For more information on the default isolation, see "Isolation level" on page 28.

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 view is a parent, 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 (SQLSTATE 23504).

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 referential constraints (other than a referential constraint with a RESTRICT delete rule), are effectively checked at the end of the statement.

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. If deleting a row in the parent table would cause a column in a dependent table to be set to null and that null value would cause the search condition of a check constraint to evaluate to false, the row is not deleted (SQLSTATE 23511).

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.

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.

The locks can prevent other application processes from performing operations on the table.

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, SQLERRD(3) in the SQLCA shows the number of rows that qualified for the delete operation. In the context of an SQL procedure statement, the value can be retrieved using the ROW_COUNT variable of the GET DIAGNOSTICS statement.

For a description of the SQLCA, see Chapter 11, “SQLCA (SQL communication area),” on page 981.

Deleting rows in a table for which row access control is active: When a DELETE statement is issued for a table for which row access control is active, the rules specified in the enabled row permissions determine whether the row can be deleted. Typically those rules are based on the authorization ID of the statement.

When multiple enabled row permissions are defined for a table, a row access control search condition is derived by application of the logical OR operator to the search condition in each enabled permission. This row access control search condition is applied to the table to determine which rows are accessible to the authorization ID of the DELETE statement. If the WHERE clause is specified in the DELETE statement, the user-specified predicates are applied on the accessible rows to determine the rows to be deleted. If there is no WHERE clause, all the accessible rows are the rows to be deleted.

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 833.

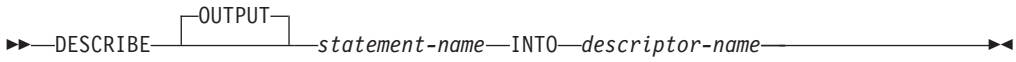
Invocation

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 "PREPARE" on page 833 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 current server.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA). For more information, see Chapter 12, "SQLDA (SQL descriptor area)," on page 985. 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 on techniques to determine the number of entries required, see "Determining how many occurrences of SQLVAR entries are needed" on page 987.

The rules for REXX are different. For more information, see Chapter 18, "Coding SQL statements in REXX applications," on page 1117.

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 SQLVAR entries for every select list item (or, column of the result table), the seventh byte is set to '2'. This technique is used in order to accommodate LOB or distinct type result columns.
- 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, the number of columns in its result table; otherwise, 0.

SQLVAR
If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR entries.

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 entries so that the first occurrence of an SQLVAR entry contains a description of the first column of the result table, the second occurrence of SQLVAR entry contains a description of the second column of the result table, and so on. For information on the values assigned to SQLVAR entries, see "Field descriptions in an occurrence of SQLVAR" on page 988.

Notes

PREPARE INTO
Information about a prepared statement can also be obtained by using the INTO clause of the PREPARE statement.

Allocating the SQLDA
In C and COBOL, 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 occurrences of SQLVAR entries are needed" on page 987 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 the number of columns in the result table. 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.

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 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.

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.

Examples

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 SQLDABC to length of 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 INPUT

The DESCRIBE INPUT statement obtains information about the input parameter markers of a prepared statement. For an explanation of prepared statements, see "PREPARE" on page 833.

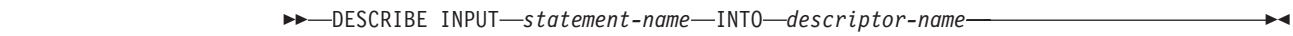
Invocation

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 "PREPARE" on page 833 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.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA). For more information, see Chapter 12, "SQLDA (SQL descriptor area)," on page 985. Before the DESCRIBE INPUT 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 INPUT statement is executed. For information on techniques to determine the number of occurrences required, see "Determining how many occurrences of SQLVAR entries are needed" on page 987.

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 entries.

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 entries so that the first occurrence of a SQLVAR entry contains a description of the first input parameter marker, the second occurrence of a SQLVAR entry contains a description of the second input parameter marker, and so on. For information on the values assigned to SQLVAR entries, see "Field descriptions in an occurrence of SQLVAR" on page 988.

Notes

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 entries 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 entries should be two times the number of input parameter markers See "Determining how many occurrences of SQLVAR entries are needed" on page 987 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 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.
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

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.

```
EXEC SQL  BEGIN DECLARE SECTION;
  char stmt1_str [200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;
struct sqlda  sqlda;
struct sqlda *sqldaPtr;

  ... /* stmt1_str  contains an INSERT statement with VALUES     */
    /* clause                                                   */
EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;

... /* code to set SQLN to five and SQLDABC to length of SQLDA */
EXEC SQL  DESCRIBE INPUT  STMT1_NAME INTO :sqlda;

...
```

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.

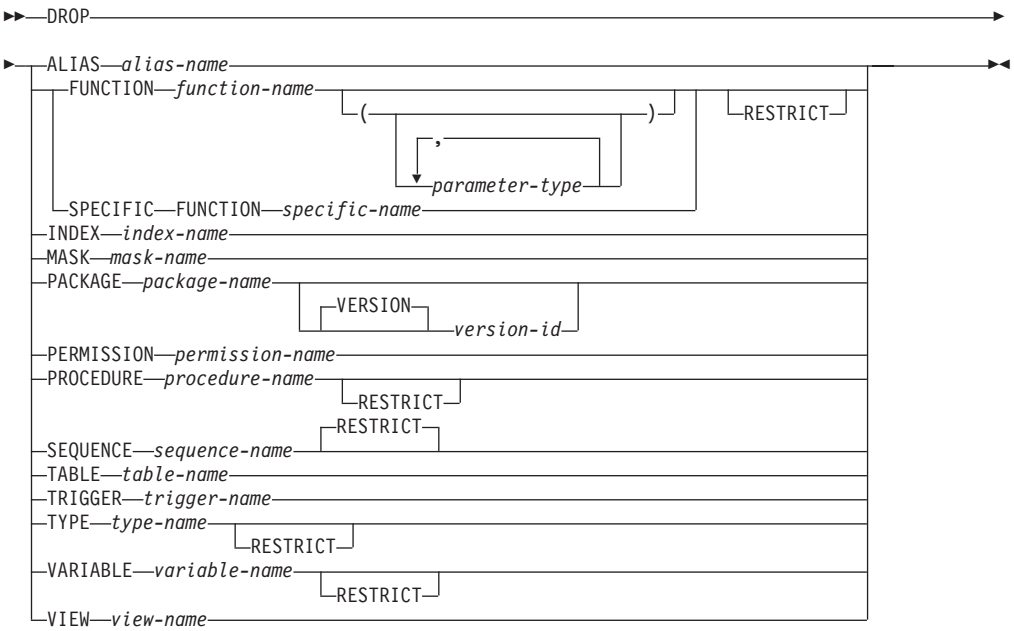
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the object
- Database administrator authority.

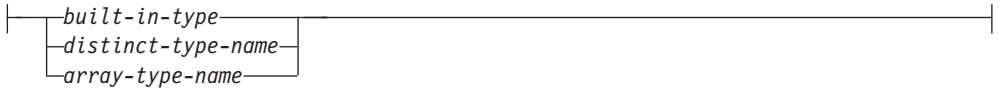
To drop an object used by a mask or permission, the privileges held by the authorization ID of the statement must include Security administrator authority.

Syntax

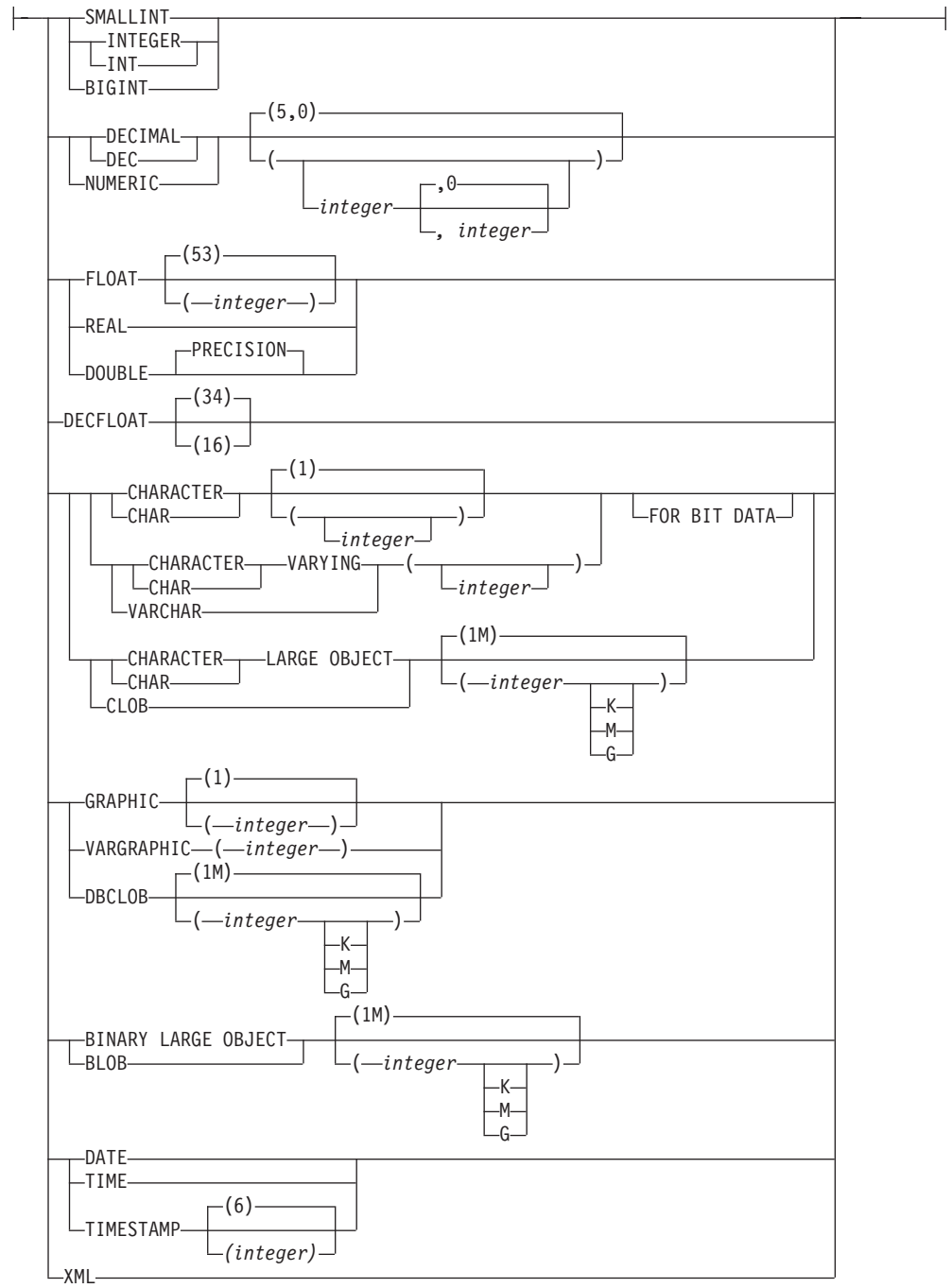


parameter-type:

data-type:



built-in-type:



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 dropped from the schema. Dropping an alias has no effect on any constraint that was defined using the alias. The effect on any tables, views, routines, or triggers that reference the alias is product-specific.

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.

The specified function is dropped from the schema. All privileges on the
G user-defined function are also dropped. The effect on any routines, triggers, or
G views that reference the function is product-specific.

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, DECIMAL() will be considered a match for a parameter of a function defined with a data type of DECIMAL(7,2). However, FLOAT cannot be specified with empty parentheses 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.

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.

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
If RESTRICT is specified, the function cannot be dropped if the function is referenced in a function, mask, materialized query table, permission, procedure, trigger, or view.

If RESTRICT is not specified, the effect on any functions, materialized query tables, procedures, triggers, or views that reference the function is product-specific.

INDEX *index-name*
Identifies the index that is to be dropped. The *index-name* must identify an index that exists at the current server, but it must not identify:

- A primary index.
- A unique index used to enforce a UNIQUE constraint.
- An index on a catalog table.

The specified index is dropped from the schema. See the product references for additional restrictions on dropping indexes.

MASK *mask-name*
Identifies the mask that is to be dropped. The *mask-name* must identify a mask that exists at the current server.

The specified mask is dropped from the schema. See the product references for additional restrictions on dropping masks.

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.

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.

PERMISSION *permission-name*
Identifies the permission that is to be dropped. The *permission-name* must identify a permission that exists at the current server.

The specified permission is dropped from the schema. See the product references for additional restrictions on dropping permissions.

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.

RESTRICT
If RESTRICT is specified, the procedure cannot be dropped if the procedure is referenced in a routine.

G If RESTRICT is not specified, the effect on any routines that reference the
G procedure is product-specific.

SEQUENCE
Identifies the sequence that is to be dropped. The *sequence-name* must identify a sequence that exists at the current server. The RESTRICT option, which is the default, prevents the sequence from being dropped if any of the following dependencies exist:

- A trigger exists such that a NEXT VALUE or PREVIOUS VALUE expression in the trigger specifies the sequence.
- An SQL function exists such that a NEXT VALUE expression in the routine body specifies the sequence.

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 it must not identify a catalog table.

The specified table is dropped from the schema. All privileges, constraints,
| indexes, masks, permissions, triggers, and views on the table are also dropped.
G Any referential constraints in which the table is the parent are dropped. The
G effect on any routines or triggers that reference the table is product-specific.

Any aliases that reference the specified table are not dropped.

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.

TYPE *type-name*
| Identifies the user-defined type that is to be dropped. The *type-name* must
| identify a user-defined type that exists at the current server.

| The specified type is dropped from the schema. All privileges on the type are
| also dropped.

| **RESTRICT**
| If RESTRICT is specified, the type cannot be dropped if the type is
| referenced in a check constraint, function, mask, materialized query table,
| permission, procedure, sequence, table, trigger, or view.

| If RESTRICT is not specified, the effect on any check constraints, functions,
| procedures, sequences, tables, triggers, or views that reference the type is
| product-specific.

| **VARIABLE** *variable-name*
| Identifies the global variable that is to be dropped. The *variable-name* must
| identify a global variable that exists at the current server.

| The specified global variable is dropped from the schema. All privileges on the
| global variable are also dropped.

RESTRICT
If RESTRICT is specified, the global variable cannot be dropped if the type is referenced in a function, mask, permission, procedure, table, trigger, or view.

If RESTRICT is not specified, the effect on any functions, mask, permission, procedures, tables, triggers, or views that reference the global variable is product-specific.

VIEW *view-name*
Identifies the view that is to be dropped. The *view-name* must identify a view that exists at the current server.

The specified view is dropped from the schema. Any view that is directly or indirectly dependent on that view is also dropped. Whenever a view is dropped, all privileges on that view are also dropped. The effect on any routines, or triggers that reference the view is product-specific.

Any aliases that reference the specified view are not dropped.

Notes

Drop effects: Whenever an object is dropped, its description is dropped from the catalog and any access plans that reference the object are invalidated. For more information, see "Packages and access plans" on page 18.

Drop restriction: In DB2 for z/OS, after an index or table is dropped, a commit must be performed before recreating an object with the same name in the default table space.

Invalidation of packages and dynamically cached statements after dropping row permissions or column masks: If row or column access control is activated on the table, dropping an enabled row permission or enabled column mask defined for that table invalidates all packages and dynamically cached statements that reference that same table.

Examples

Example 1: Drop the table named MY_IN_TRAY.

```
DROP TABLE MY_IN_TRAY
```

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 TYPE DOCUMENT RESTRICT
```

Example 5: Assume that ATOMIC_WEIGHT is the only function with that name in schema CHEM. Drop ATOMIC_WEIGHT.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT
```

Example 6: Drop the function named CENTER, using the function signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTER (INTEGER, DOUBLE)
```

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 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 described in "BEGIN DECLARE SECTION" on page 582.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.

Examples

See "BEGIN DECLARE SECTION" on page 582 for examples that use the END DECLARE SECTION statement.

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

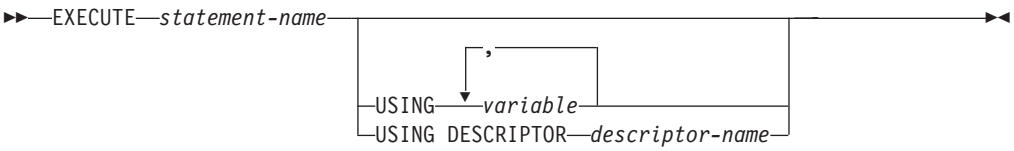
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See "PREPARE" on page 833 for the authorization required to create a prepared statement.

Syntax



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 833.) 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 or more host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and variables. A global variable must not be specified. 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.

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 (Note that the rules for REXX are different. For more information, see Chapter 18, "Coding SQL statements in REXX applications," on page 1117):

- SQLN to indicate the number of SQLVAR entries 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 entries to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all occurrences of SQLVAR entries. If an SQLVAR entry includes a LOB or distinct type based on a LOB, there must be additional SQLVAR entries for each parameter. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR entries, see Chapter 12, "SQLDA (SQL descriptor area)," on page 985.

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.

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 55 on page 838.

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 77. Thus:

- V must be compatible with the target.
- If V is a string, its length must not be greater than the length attribute of the target.
- If V is a number, the whole part of the number must not be truncated.
- 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, V must not be null.

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.

Examples

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) BINARY.
  77 TIM              PIC S9(3)V9(2).
  01 HOLDER.
     49  HOLDER-LENGTH    PIC S9(4) BINARY.
     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

The EXECUTE IMMEDIATE statement:

- Prepares an executable form of an SQL statement from a character string form of the statement
- Executes the SQL statement

EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements. It can be used to prepare and execute SQL statement that contain neither variables nor parameter markers.

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:

- The READ privilege on the global variable
- Database administrator authority

The authorization rules are those defined for the SQL statement specified by EXECUTE IMMEDIATE. For example, see "INSERT" on page 810 for the authorization rules that apply when an INSERT statement is executed using EXECUTE IMMEDIATE.

The authorization ID is the run-time authorization ID.

Syntax

Description

variable
Identifies a variable that must be described in accordance with the rules for declaring character-string variables. The variable must not have a CLOB data type, and an indicator variable must not be specified. A global variable must not be specified.

In COBOL it must be a varying-length string variable. In C, it must be the VARCHAR structured form of a string variable rather than the NUL-terminated form.

The value of the identified variable is called the *statement string*.

The statement string must be one of the following SQL statements:[121]

121. A select-statement is not allowed. To dynamically process a select-statement, use the PREPARE, DECLARE CURSOR, and OPEN statements.

ALTER	INSERT	SAVEPOINT
COMMENT	LOCK TABLE	SET CURRENT DECFLOAT ROUNDING MODE
COMMIT	MERGE	SET CURRENT DEGREE
CREATE	REFRESH TABLE	SET ENCRYPTION PASSWORD
DECLARE GLOBAL TEMPORARY TABLE	RELEASE SAVEPOINT	SET PATH
DELETE	RENAME	SET SCHEMA
DROP	REVOKE	TRUNCATE
GRANT	ROLLBACK	UPDATE

The statement string must not:

- Begin with EXEC SQL.
- End with END-EXEC or a semicolon.
- Include references to variables. Global variables are allowed.
- 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 invalid, it is not executed and the error is returned. If the SQL statement is valid, but an error occurs during its execution, that error is returned.

Notes

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.

Examples

Use C to execute the SQL statement in the variable Qstring.

```
EXEC SQL INCLUDE SQLCA;
void main ()
  {

    EXEC SQL BEGIN DECLARE SECTION;

    char Qstring[100] =
      "INSERT INTO WORK_TABLE SELECT * FROM EMPPROJACT WHERE ACTNO >= 100";

    EXEC SQL END DECLARE SECTION;

  .
  .
  .
    EXEC SQL EXECUTE IMMEDIATE :Qstring;

    return;
  }
```

FETCH

The FETCH statement positions a cursor on the next row of its result table and assigns the values of 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.

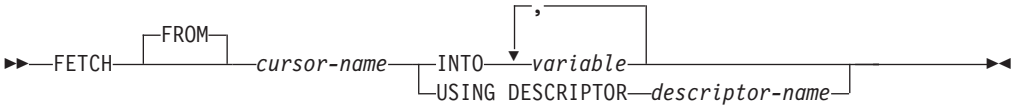
Authorization

If a global variable is referenced in the INTO clause, the privileges held by the authorization ID for the statement must include at least one of the following:

- The WRITE privilege on the global variable
- Database administrator authority

See "DECLARE CURSOR" on page 749 for an explanation of the authorization required to use a cursor.

Syntax



Description

cursor-name
Identifies the cursor to be used in the fetch operation. The *cursor-name* must identify either a declared cursor as explained in "DECLARE CURSOR" on page 749 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.

INTO *variable*,...
Identifies one or more host structures or variables that must be described in accordance with the rules for declaring host structures and variables. In the operational form of INTO, a reference to a 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.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more output variables.

Before the FETCH statement is processed, the user must set the following fields in the SQLDA (note that the rules for REXX are different, for more information see Chapter 18, "Coding SQL statements in REXX applications," on page 1117):

- 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*(N), where N 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 Chapter 12, "SQLDA (SQL descriptor area)," on page 985.

The USING DESCRIPTOR clause is not supported for a FETCH statement within a Java program.

Notes

Cursor position: An open cursor has three possible positions:

- Before a row
- On a row
- After the last row.

If the cursor is currently positioned on or after the last row of the result table:

- SQLSTATE is set to '02000'.
- The cursor is positioned after the last row.
- Values are not assigned to variables.

If the cursor is currently positioned before a row, the cursor is positioned on that row, and the values of that row are assigned to variables as specified by INTO or USING.

If the cursor is currently positioned on a row other than the last row, the cursor is positioned on the next row and values of that row are assigned to variables as specified by INTO or USING.

If a cursor is 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 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 77. 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 and the *variable* is a host variable or a descriptor is specified, an indicator variable must be provided. If an assignment error occurs, the values in the variables are unpredictable.

Result column evaluation considerations: If an error occurs as the result of an arithmetic expression in the select list of an outer SELECT statement (such as 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, if the *variable* is a host variable or a descriptor is specified, an indicator variable must be provided. The value of the variable is undefined. In this case, however, the indicator variable (if any) is set to -2. Processing of the statement continues and a warning is returned. If the *variable* is a host variable or a descriptor is specified and 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 occurs. [122]

If the specified variable is not large enough to contain the result, a warning is returned (SQLSTATE 01004) 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 a CLOB, DBCLOB or BLOB value is truncated, the length may not be returned in the indicator variable.

It is possible that a warning may not be returned on a FETCH. This occurs as a result of optimizations such as the use of system temporary tables or blocking. It is also possible that the returned warning applies to a previously fetched row.

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 an error is returned. See "Datetime assignments" on page 83 for details.

Examples

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
```

122. In DB2 for LUW, the database configuration parameter dft_sqlmathwarn must be set to yes for this behavior to be supported.

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 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, or a double-byte character large object 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 FETCH, SELECT INTO, assignment statement, 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 is specified and an error is returned on one of the locators, it is possible that some locators have been freed and others have not been freed.

Examples

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that the 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.
```

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

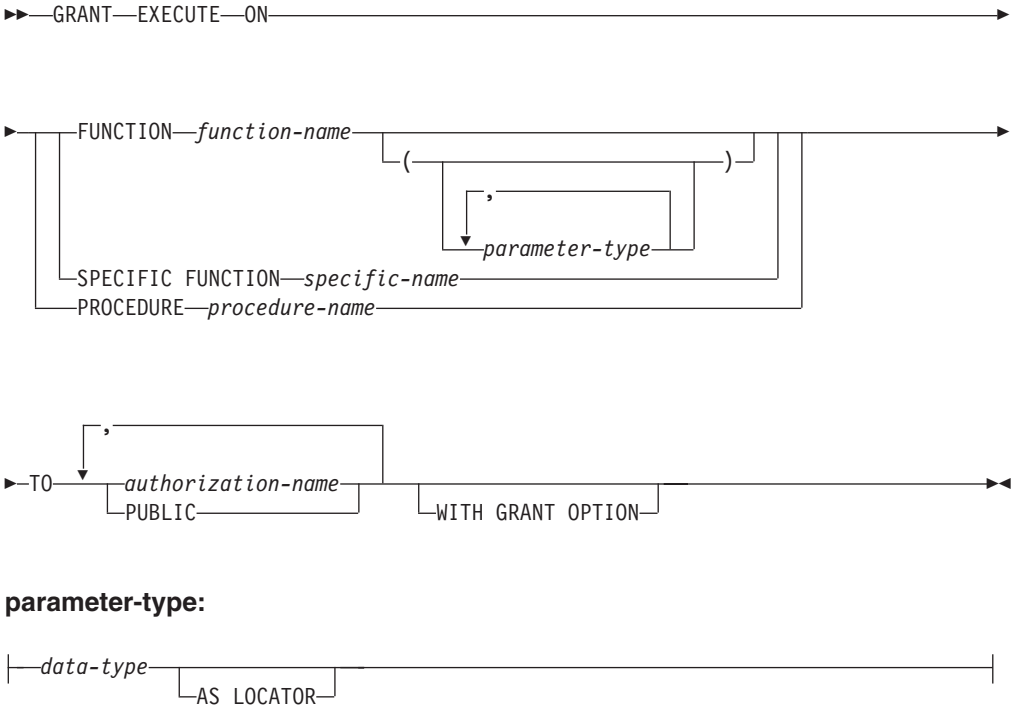
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the function or procedure
- The EXECUTE privilege on the function or procedure with the WITH GRANT OPTION
- Security administrator 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
- Security administrator authority.

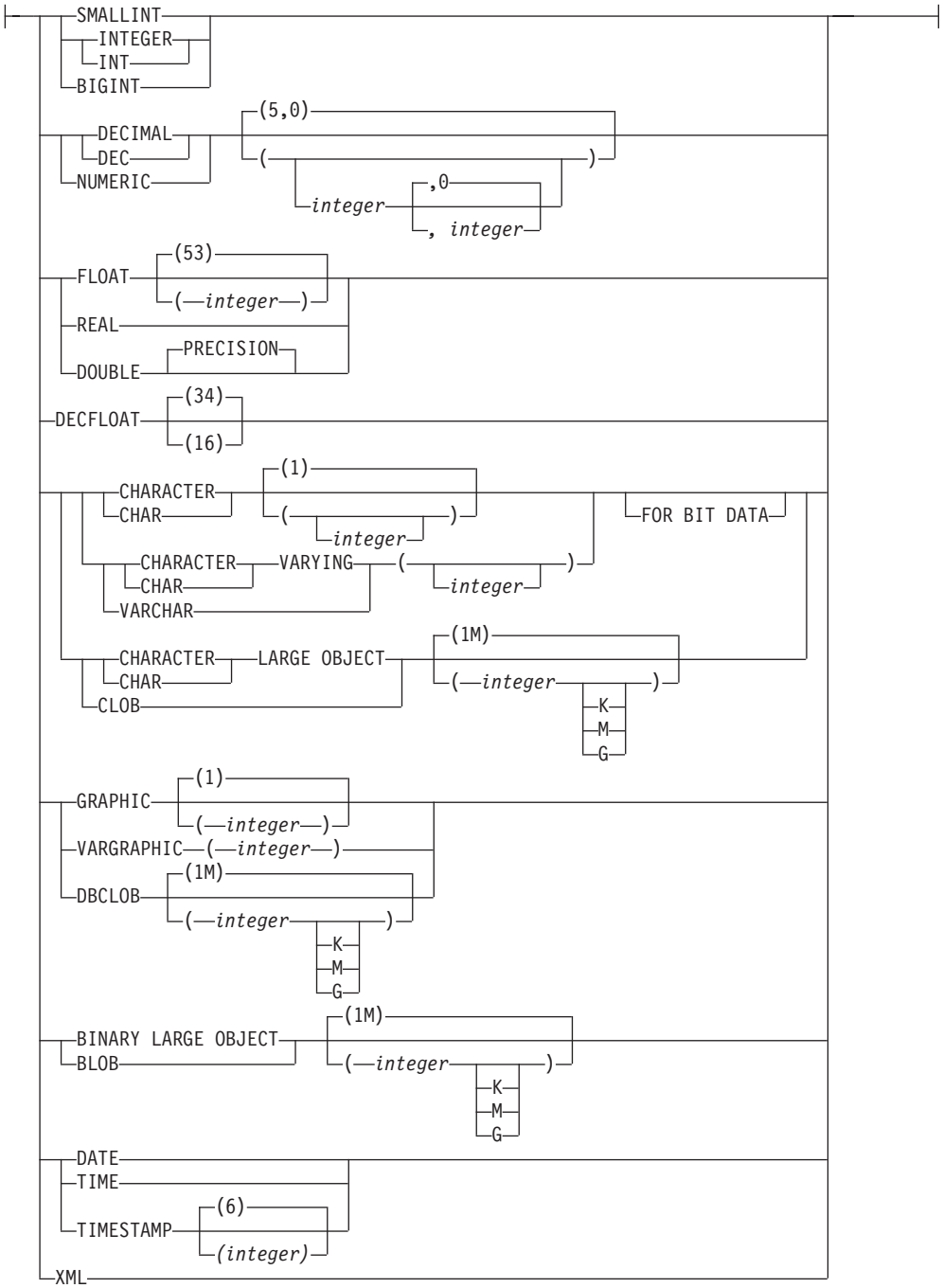
Syntax



parameter-type:

data-type:

built-in-type:



Description

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. The function can be identified by 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 user-defined type name is specified, the database manager searches the SQL path to resolve the schema name for the user-defined 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, DECIMAL() will be considered a match for a parameter of a function defined with a data type of DECIMAL(7,2). However, FLOAT cannot be specified with empty parentheses 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 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.

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.

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 *procedure-name*
Identifies the procedure on which the privilege is granted. The *procedure-name* must identify a procedure that exists at the current server.

TO Indicates to whom the privilege is granted.

authorization-name,...
Lists one or more authorization IDs.[123]

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 21.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the EXECUTE privilege to others users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the EXECUTE privilege to others unless they have received that authority from some other source.

Notes

Built-in functions: Privileges cannot be granted on built-in functions.

Examples

Grant the EXECUTE privilege on procedure PROCA to PUBLIC.

```
GRANT EXECUTE
  ON PROCEDURE PROCA
  TO PUBLIC
```

123. In DB2 for z/OS, the CURRENT RULES special register must be set to 'STD' to grant privileges to the authorization ID of the GRANT statement itself.

GRANT (package privileges)

This form of the GRANT statement grants the privilege to execute statements in 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

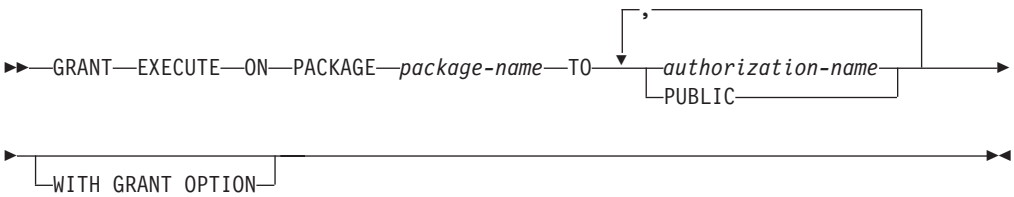
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the package
- The EXECUTE privilege on the package with the WITH GRANT OPTION
- Security administrator 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
- Security administrator authority.

Syntax



Description

EXECUTE
Grants the privilege to execute SQL statements in a package.

ON PACKAGE *package-name*
Identifies the package on which the EXECUTE privilege is granted. The *package-name* must identify a package that exists at the current server.

TO Indicates to whom the privilege is granted.

authorization-name,...
Lists one or more authorization IDs. In DB2 for LUW, the authorization ID of the GRANT statement itself cannot be specified.[124]

124. In DB2 for z/OS, the CURRENT RULES special register must be set to 'STD' to grant privileges to the authorization ID of the GRANT statement itself.

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 21.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the EXECUTE privilege to others users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the EXECUTE privilege to others unless they have received that authority from some other source.

Examples

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

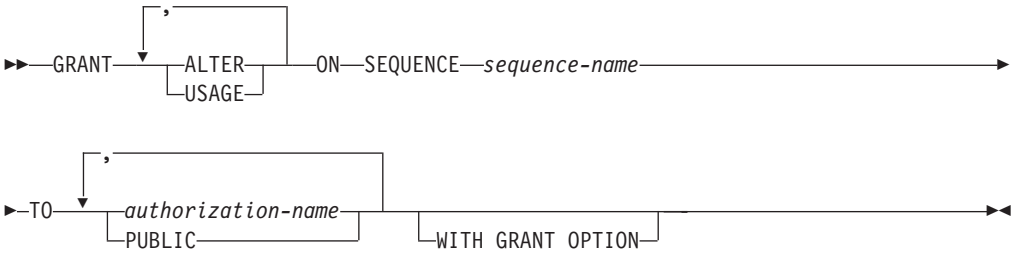
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the sequence
- The WITH GRANT OPTION for at least one of the specified privileges.
- Security administrator 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
- Security administrator authority.

Syntax



Description

ALTER
Grants the privilege to use the ALTER SEQUENCE statement 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 privilege is granted.

authorization-name,...
Lists one or more authorization IDs.[125]

125. In DB2 for z/OS, the CURRENT RULES special register must be set to 'STD' to grant privileges to the authorization ID of the GRANT statement itself.

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 21.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the privileges to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the privileges to others unless they have received that authority from some other source.

Examples

Grant 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 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

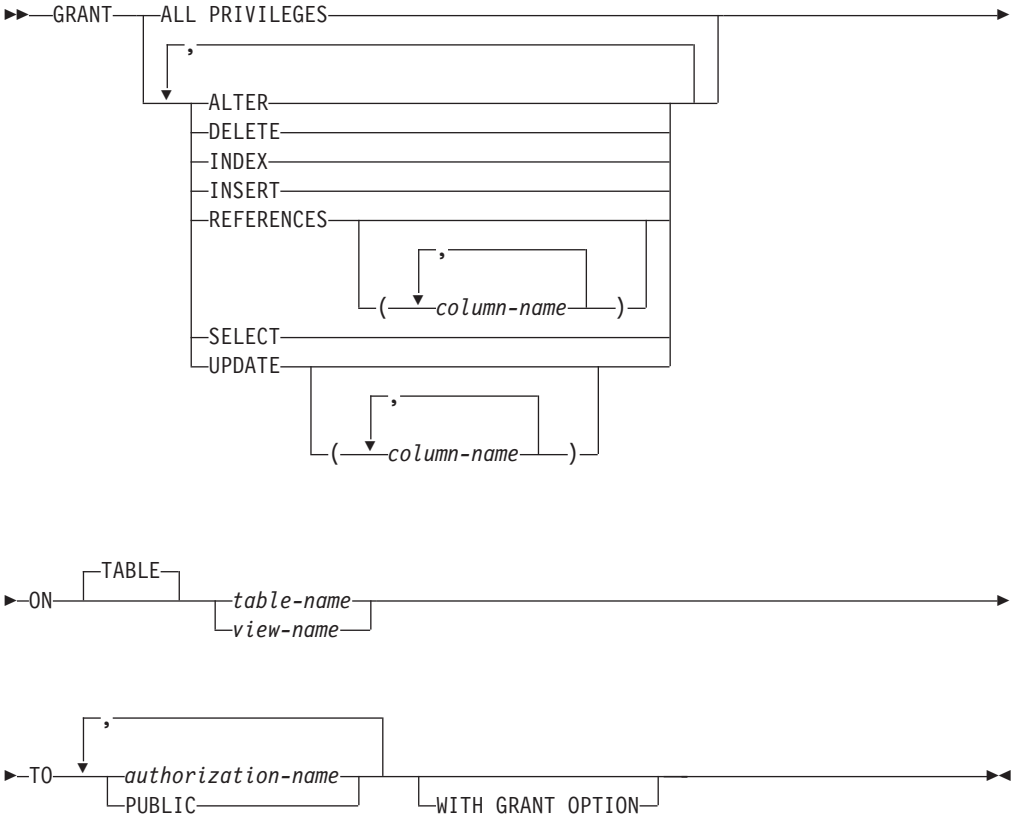
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the table or view
- The WITH GRANT OPTION for at least one of the specified privileges. If ALL is specified, the authorization ID must have some grantable privilege on the table or view
- Security administrator 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
- Security administrator authority.

Syntax



Description

ALL PRIVILEGES
Grants one or more privileges on the specified table or view. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the identified table or view.

ALTER
Grants the privilege to alter the specified table or create a trigger on the specified table. This privilege cannot be granted on a view.

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 the specified table is a parent. If a list of column names is not specified or if REFERENCES is granted via the specification of ALL PRIVILEGES, the grantee(s) can define referential constraints using all columns of the table as a parent key, even those added later via the ALTER TABLE statement. This privilege cannot be granted on a view.

REFERENCES (*column-name***,...)**
Grants the privilege to add a referential constraint in which the 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 the table identified in the ON clause. This privilege cannot be granted on 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 column names is not specified or if UPDATE is granted via the specification of ALL PRIVILEGES, the grantee(s) can update all updatable columns of the table or view, 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 for the specified table or view 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 the table or view identified 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 table or view 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. [126]

PUBLIC
Grants the privilege(s) to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 21.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the privileges to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the privileges to others unless they have received that authority from some other source.

Notes

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
```

126. In DB2 for z/OS, the CURRENT RULES special register must be used to grant privileges to the authorization ID of the GRANT statement itself.

GRANT (type privileges)

This form of the GRANT statement grants privileges on a user-defined type.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

G In DB2 for LUW, this statement is not supported. Instead, PUBLIC implicitly has
G the USAGE privilege on all user-defined types.

Authorization

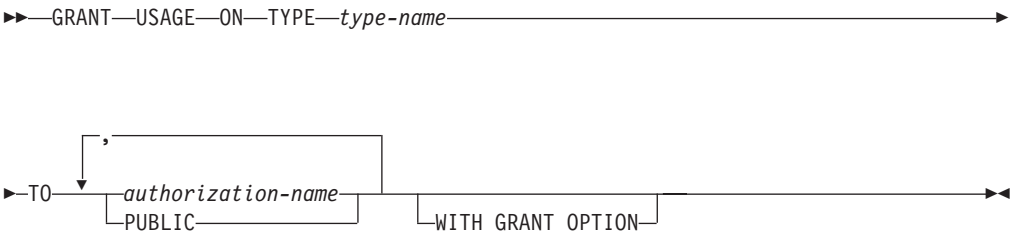
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the user-defined type
- The USAGE privilege on the user-defined type with the WITH GRANT OPTION
- Security administrator 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 user-defined type
- Security administrator authority.

Syntax



Description

USAGE
Grants the privilege to use the user-defined type in tables, functions, procedures, or CAST expressions.

ON TYPE *type-name*
Identifies the user-defined type on which the privilege is granted. The *type-name* must identify a user-defined type that exists at the current server.

TO Indicates to whom the privilege is granted.

authorization-name,...
Lists one or more authorization IDs.[127]

127. In DB2 for z/OS, the CURRENT RULES special register must be set to 'STD' to grant privileges to the authorization ID of the GRANT statement itself.

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 21.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the USAGE privilege to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the USAGE privilege to others unless they have received that authority from some other source.

Notes

Cast function implications: The GRANT (type privileges) statement does not grant a user the privilege to execute the cast functions that are associated with the user-defined type. The GRANT (Function or Procedure Privileges) statement must be used to grant the EXECUTE privilege to the cast functions associated with the user-defined type.

Examples

Grant the USAGE privilege on user-defined type SHOESIZE to user JONES.

```
GRANT USAGE
  ON TYPE SHOESIZE
  TO JONES
```

GRANT (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

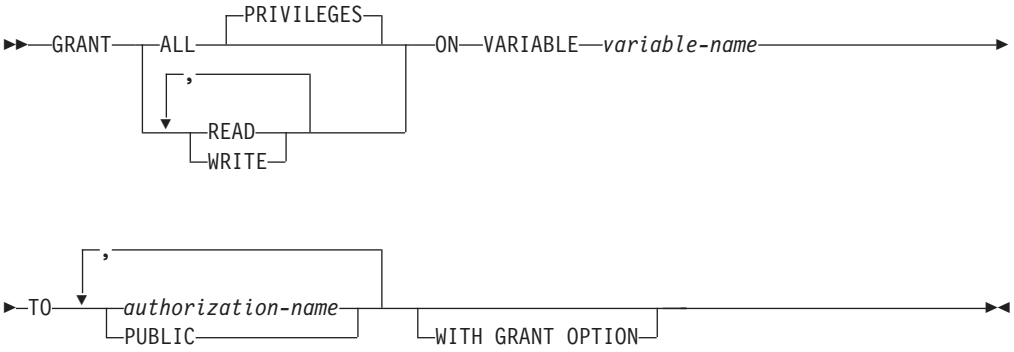
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the variable
- The WITH GRANT OPTION for at least one of the specified privileges.
- Security administrator 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 variable
- Security administrator 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 variable.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

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 privilege is granted.

authorization-name,...
Lists one or more authorization IDs.[128]

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership" on page 21.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the privileges to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the privileges to others unless they have received that authority from some other source.

Examples

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
```

128. In DB2 for z/OS, the CURRENT RULES special register must be set to 'STD' to grant privileges to the authorization ID of the GRANT statement itself.

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

None required.

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 SQLSTATE or stand-alone SQLCODE. In COBOL, INCLUDE SQLCA can only be specified within the WORKING-STORAGE SECTION. If INCLUDE SQLCA is not specified in C or COBOL, then the variable SQLSTATE or SQLCODE 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 526.

For a description of the SQLCA, see Chapter 11, "SQLCA (SQL communication area)," on page 981.

SQLDA
Indicates the description of an SQL descriptor area (SQLDA) is to be included. INCLUDE SQLDA can be specified in C and COBOL programs. In COBOL, INCLUDE SQLDA can only be specified within the WORKING-STORAGE SECTION.

For a description of the SQLDA, see Chapter 12, "SQLDA (SQL descriptor area)," on page 985.

name
Identifies an external file or member containing text that is to be included in the source program being precompiled. In COBOL, INCLUDE *name* must not be specified in other than the DATA DIVISION or PROCEDURE DIVISION.

G The rules for forming the name and the technique used to map the name to an
G external file or library member are product-specific.

The included text can contain any statements of the host language and any SQL statements other than INCLUDE statements.

When a program is precompiled, the INCLUDE statement is replaced by source statements.

The INCLUDE statement must be specified at a point in a program where its source statements are allowed.

Examples

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 inserts the row into the table on which the view is based if no INSTEAD OF INSERT 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 *INSERT using VALUES* form is used to insert a single row 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 the result of the query.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

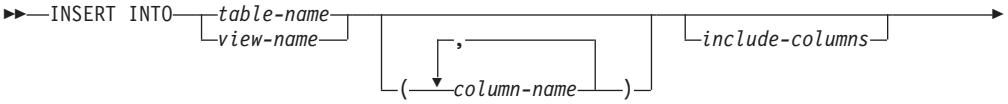
The privileges held by the authorization ID of the statement must include at least one of the following:

- The INSERT privilege for the table or view
- Ownership of the table [129]
- Database administrator authority.

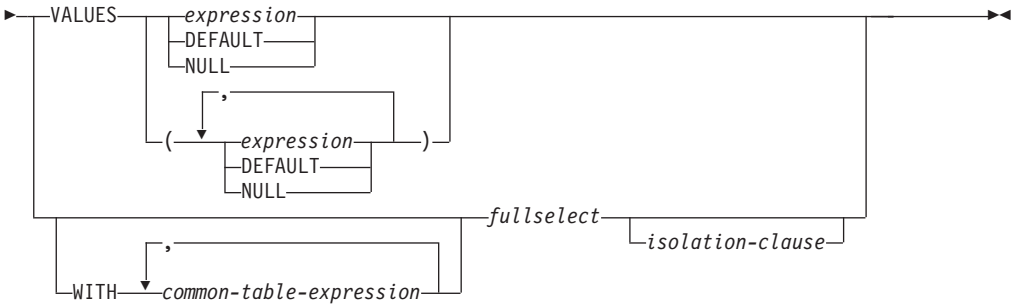
If a *fullselect* is specified, the privileges held by the authorization ID of the statement must also include at least one of the following:

- For every table or view identified in the fullselect:
  - The SELECT privilege on the table or view, or
  - Ownership of the table or view.
- Database administrator authority.

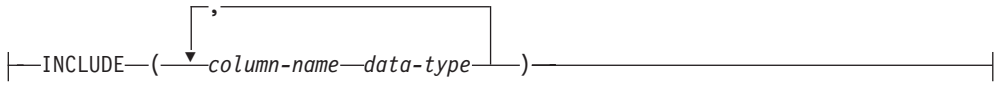
Syntax



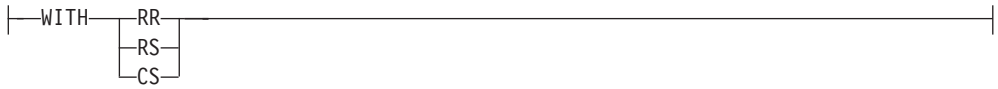
129. The INSERT privilege on a view is only inherent in database administrator authority. Ownership of a view does not necessarily include the INSERT privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.



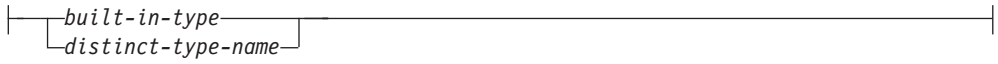
include-columns:



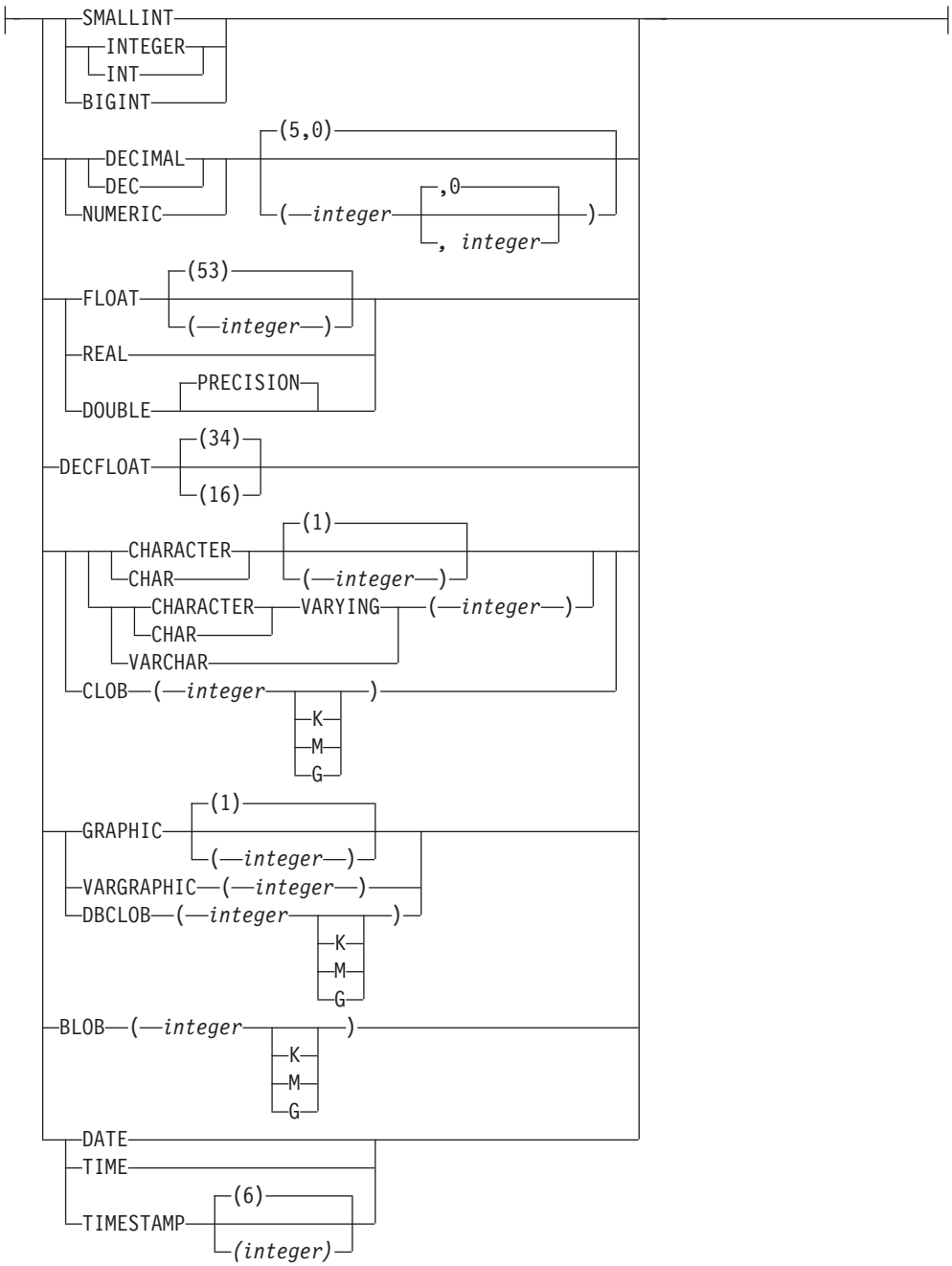
isolation-clause:



data-type:



built-in-type:



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 743.

(*column-name***,...)**
Specifies the columns for which insert values are provided. Each name must identify 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 743.

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.

In all the products, SQL statements can be implicitly or explicitly rebound (prepared again). The effect of a rebind on INSERT statements that do not include a column list is as follows:

- In DB2 for z/OS, and DB2 for LUW, the implicit list of names is reestablished. Therefore, the number of columns into which data is inserted may change.
- In DB2 for i, the list of names is not reestablished. Therefore, the number of columns into which data is inserted by the statement does not change.

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*.

data-type
Specifies the data type of the include column. For a description of *data-type*, see "CREATE TABLE" on page 688.

VALUES
Specifies one new row in the form of a list of values. Each variable in the clause must identify a host structure or variable that is declared in the program in accordance with the rules for declaring host structures and variables. In the operational form of the statement, a reference to a structure is replaced by a reference to each of its variables. For further information on variables and structures, see "References to host variables" on page 116 and "Host structures" on page 121.

The number of values 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 into the first column in the list, then the second value into the second column, and so on.

expression
Any expression of the type described in "Expressions" on page 130, that does not include a column name. If *expression* is a *variable*, the host 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 688).
- 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 an 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 an identity column or row change timestamp column defined as GENERATED ALWAYS. For information on default values of data types, see the description of the DEFAULT clause for CREATE TABLE in "CREATE TABLE" on page 688.

NULL
Specifies the null value as the value of the column. Specify NULL only 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 506.

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 500.

When the base object of the INSERT and the base object of the fullselect or any subquery of the fullselect, are the same table, the fullselect 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, then the second value in the second column, and so on.

G In DB2 for z/OS, if the object table is self-referencing, the fullselect must not
G return more than one row.

isolation-clause
Specifies the isolation level used by the statement.

WITH

Introduces the isolation level, which may be one of:
- *RR* Repeatable read
- *RS* Read stability
- *CS* Cursor stability

If *isolation-clause* is not specified, the default isolation is used. See "Isolation level" on page 28 for a description of how the default is determined.

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 the insert is 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 "Assignments and comparisons" on page 77.

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.

- *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. In the case of a multiple-row INSERT statement, this would occur after all rows were 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).

  All check constraints are effectively validated 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 743.

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 return 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 be equal to some value of the parent key of the parent table in the relationship (SQLSTATE 23503).

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.

XML values: A value that is inserted into an XML column must be a well-formed XML document.

Inserting rows into a table for which row or column access control is enforced: When an INSERT statement is issued for a table for which row or column access control is enforced, the rules specified in the enabled row permissions or column masks determine whether the row can be inserted. Typically those rules are based on the authorization ID of the statement. The following rules describe how the enabled row permissions and column masks are used during INSERT:

- A row to be inserted must not be effected by enabled column masks whose columns are referenced while deriving the source values for the row.

  When a column is referenced while deriving the values of a new row, if the column has an enabled column mask, the masked value is used to derive the new values. If the object table also has column access control activated, the column mask that is applied to derive the new values must ensure that the evaluation of the access control rules defined in the column mask resolves the column to itself, not to a constant or to an expression. If the column mask does not mask the column to itself, the new value cannot be used for insert and an error is returned at run time.

  If the OVERRIDING USER VALUE clause is specified, the corresponding values in the new row are ignored, and the above rule for column masks is not applicable to those values.
- If the row can be inserted, and there is a BEFORE INSERT trigger for the table, the trigger is activated.

  Within the trigger actions, the new values for insert can be modified in the transition variables. When the values return from the trigger, the final values for the new values are the ones for insert.
- A row to be inserted must conform to the enabled row permissions.

  When multiple enabled row permissions are defined for a table, a row access control search condition is derived by application of the logical OR operator to the search condition in each enabled row permission. A row that conforms to the enabled row permissions is a row that if the row is inserted it can be retrieved back using the row access control search condition.

  Column masks are not applicable in this process.
- If the rows can be inserted, and there is an AFTER INSERT trigger for the table, the trigger is activated.

The preceding rules are not applicable to the *include-columns*. The *include-columns* are subject to the rules for the select list because they are not the columns of the object table of the INSERT statement.

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 the INSERT statement, changes from this statement and any triggered SQL statements are rolled back.

Number of rows inserted: When an INSERT statement is completed, SQLERRD(3) in the SQLCA SQLCA indicates the number of rows that were passed to the insert operation. In the context of an SQL procedure statement, the value can be retrieved using the ROW_COUNT variable of the GET DIAGNOSTICS statement.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful INSERT statement. Until these 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 isolation level UR through a read-only cursor, SELECT INTO statement, or subquery.

The locks can prevent other application processes from performing operations on the table.

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.

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: 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
```

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:

- The SELECT privilege for the table
- Ownership of the table
- Database administrator 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 it 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.

IN EXCLUSIVE MODE
Prevents concurrent application processes from executing any operations on the table. This may or may not apply to concurrent application processes running at isolation level UR. The rule is product-specific.

Notes

Locks obtained: Locking is used to prevent concurrent operations. A lock is not necessarily acquired during the execution of the LOCK TABLE statement if a suitable lock already exists. The lock that prevents concurrent operations is held until the end of the unit of work.

Examples

Request an exclusive lock on the DEPARTMENT table.

```
LOCK TABLE DEPARTMENT IN EXCLUSIVE MODE
```

MERGE

The MERGE statement updates a target (a table or view) using the specified input data. Rows in the target that match the input data may be updated as specified, and rows that do not exist in the target may be inserted as specified. Updating or inserting a row in a view updates or inserts the row into the tables on which the view is based.

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:

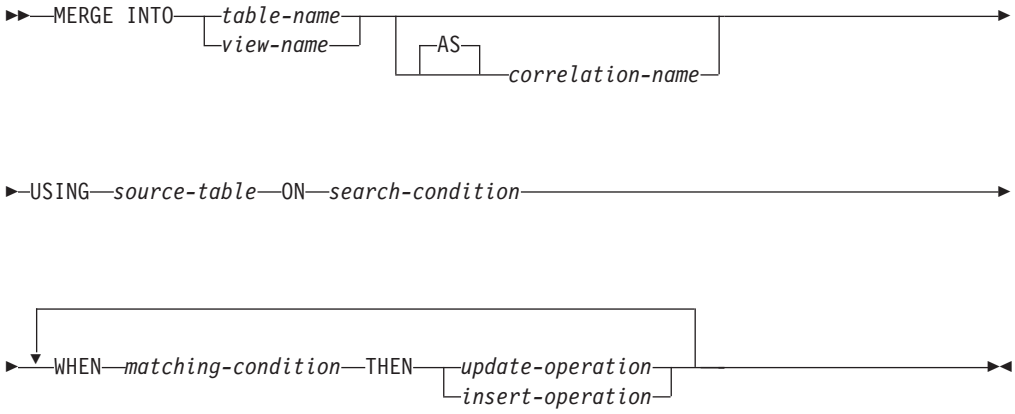
- If an insert operation is specified, the INSERT privilege on the table or view. If an update operation is specified:
  - The UPDATE privilege on the table or view or
  - The UPDATE privilege on each column to be updated

  or
- Database administrator authority.

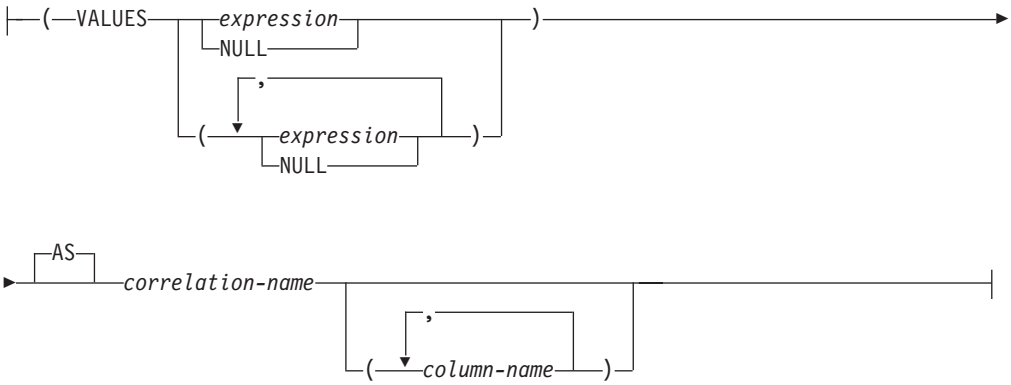
If *search-condition*, *insert-operation*, or *assignment-clause* includes a fullselect, the privileges held by the authorization ID of the statement must also include at least one of the following:

- For each table or view identified in the *fullselect*, the SELECT privilege on the table or view or
- Database administrator authority.

Syntax



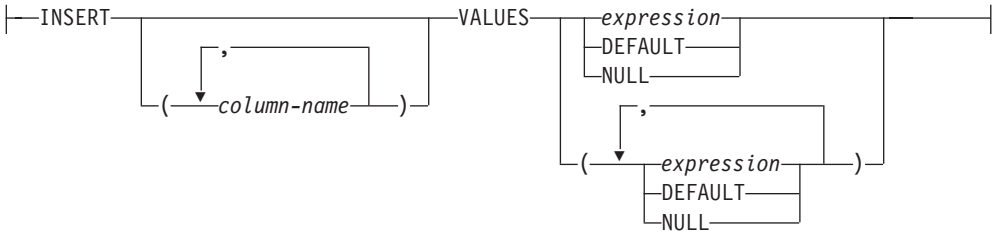
source-table:



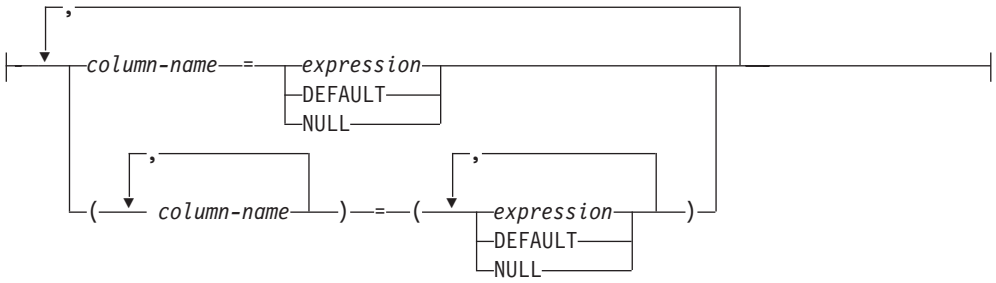
matching-condition:

update-operation:

insert-operation:



assignment-clause:



Description

table-name **or** *view-name*

Identifies the target of the update or insert 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 view that directly or indirectly contains a WHERE clause that references a subquery or a routine defined with NOT DETERMINISTIC or EXTERNAL ACTION.

If a view is specified as the target of the MERGE statement, the view must not be defined with any INSTEAD OF triggers.

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.

USING *source-table*
Specifies the row to be merged into the target.

expression
Specifies an expression of the type that is described in "Expressions" on page 200. The expression must not include a column name. The expression must not reference a NEXT VALUE or PREVIOUS VALUE expression. Any variable that is specified must be described in the application program according to the rules for declaring variables.

If the expression is a variable, or if a variable is being explicitly cast, the variable can include an indicator variable. Indicator variables can be enabled for extended indicator variables.

NULL
Specifies a null value.

AS *correlation-name*
Specifies a correlation name for the *source-table*.

column-name
Specifies a column name to associate the input data to the SET *assignment-clause* for an update operation or the VALUES clause for an insert operation.

ON *search-condition*
Specifies the predicates used to determine whether the 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 *source-table*. A subquery is not allowed in the *search-condition*. If a *column-name* exists in both the target and the *source-table*, the column name must be qualified.

For the row of the *source-table*, the *search-condition* is applied to each row of the target. If the *search-condition* is evaluated as true and the target is not empty, the specified WHEN MATCHED clause is used. Otherwise, the specified WHEN NOT MATCHED clause is used.

The *search-condition* cannot contain expressions that use aggregate functions or non-deterministic scalar functions.

WHEN *matching-condition*
Specifies the condition under which the *update-operation* or *insert-operation* 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 can be specified after THEN. WHEN MATCHED must not be specified more than one time.

NOT MATCHED
Indicates the operation to be performed on the rows where the ON

search-condition is false or unknown. Only INSERT can be specified after THEN. WHEN NOT MATCHED must not be specified more than one time.

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.

When extended indicator variables are enabled, a column of the source table must not be referenced multiple times in a single *modification-operation*.

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.

expression
Indicates the new value of the column. The expression is any expression of the type that is described in "Expressions" on page 200. The expression must not include an aggregate function.

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 the a column is referenced in the expression and the same column name exists in both the target table and the source table, the column name must be qualified .

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 688.

DEFAULT must be specified for a column that was defined as GENERATED ALWAYS. 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.

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 743.

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 116.

expression
An *expression* of the type described in "Expressions" on page 130, 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 a 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 688.

DEFAULT must be specified for a column that was defined as GENERATED ALWAYS. 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.

MERGE Rules

- Only one *update-operation* and one *insert-operation* 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 the row in the *source-table*. 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 or INSERT statement.

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, function, or a subselect.

Notes

Logical order of processing: MERGE is an atomic statement. The logical order of processing is:

1. Determine the set of rows to be processed from the 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. Apply each *update-operation* or *insert-operation* to the applicable rows in the order of specification. The triggers activated by each *update-operation* or *insert-operation* are executed. Statement level triggers are activated even if no rows satisfy the *update-operation* or *insert-operation*. Each *update-operation* or *insert-operation* can affect the triggers of each subsequent *update-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. The ROW_COUNT item and SQLERRD(3) does not include the number of rows that were operated on as a result of triggers.

For a description of ROW_COUNT, see "GET DIAGNOSTICS statement" on page 935. For a description of the SQLCA, see Chapter 11, "SQLCA (SQL communication area)," on page 981.

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* 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-operations* or *insert-operations* 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: Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful MERGE statement. Until these locks are released by a commit or rollback operation, an updated row can only be accessed by:

- the application process that performed the update,
- another application process using isolation level UR through a read-only cursor, a SELECT INTO statement, or a subquery.

The locks can prevent other application processes from performing operations on the table.

Tables with active row and column access controls: For information about how enabled row permissions and column masks affect the update and insert operations in the MERGE statement, see the INSERT and UPDATE statement information.

Examples

Example 1: 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;
```

OPEN

The OPEN statement opens a cursor so that it can be used to fetch rows from its result table.

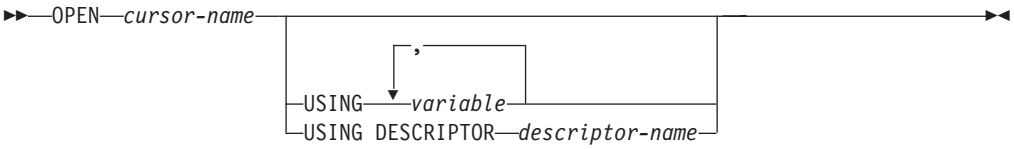
Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See "DECLARE CURSOR" on page 749 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 "Notes" on page 751. 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 was not successfully prepared, or is not a *select-statement*, an error is returned.

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 either be derived during the execution of the OPEN statement (in which case a temporary table is created for 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 state of the cursor is effectively 'after the last row.' An empty table does not cause an SQLSTATE warning of '02000' when the OPEN statement is executed. A subsequent fetch for the cursor may return the SQLSTATE warning of '02000'.

USING

Introduces the values that are substituted for the parameter markers (question marks) or variables in the statement of the cursor. For an explanation of parameter markers, see "PREPARE" on page 833.

- 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 .

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 global variable must not be specified. 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.

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:

- SQLN to indicate the number of SQLVAR entries 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 entries to indicate the attributes of the variables.

Note that the rules for REXX are different. For more information see Chapter 18, “Coding SQL statements in REXX applications,” on page 1117.

The SQLDA must have enough storage to contain all occurrences of SQLVAR entries. If an SQLVAR entry includes a LOB or distinct type based on a LOB, there must be additional SQLVAR entries for each parameter. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR entries, see Chapter 12, “SQLDA (SQL descriptor area),” on page 985.

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.

Notes

Closed state of cursors: Cursors are in an open state after a successful OPEN statement. The state of the cursor becomes closed in many ways:

- All cursors in a program are in a closed state when the program is first called.
- Cursors declared not using the WITH HOLD option are in a closed state when a unit of work is committed.
- Cursors declared in a procedure not using the WITH RETURN clause may be closed when the procedure returns. For more information, see “DECLARE CURSOR” on page 749.
- Cursors are in a closed state when a unit of work is rolled back.

A cursor can also be in the closed state because:

- A CLOSE statement was executed.
- An error was detected that made the position of the cursor unpredictable.
- 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, the database manager 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 the following ways:

- An error can occur during OPEN that would otherwise not occur until some later FETCH statement.
- 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 the value of 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 55 on page 838.

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 77. Thus:

- V must be compatible with the target.

- If V is a string, its length (including trailing blanks) must not be greater than the length attribute of the target.
- If V is a number, the whole part of the number must not be truncated.
- 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, V must not be null.

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.

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 CUR 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 CUR before the first row to be fetched.

```
EXEC SQL  DECLARE CUR CURSOR FOR
            SELECT DEPTNO, DEPTNAME, MGRNO FROM DEPARTMENT
              WHERE ADMRDEPT = 'A00'  END-EXEC.

EXEC SQL  OPEN CUR  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 long hv_int;
   char hv_vchar64[65];
   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 2, but in this case, the number and data types of the parameter markers in the WHERE clause 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 INTO :sqlda;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

/* Set up the SQLDA */

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*.

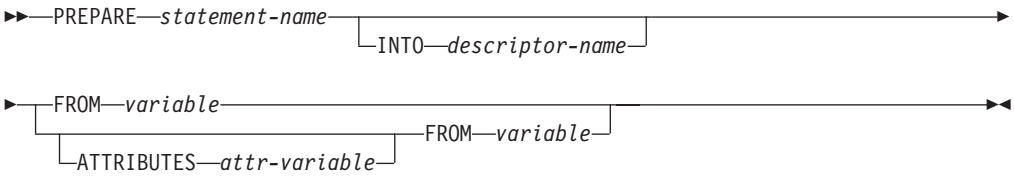
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

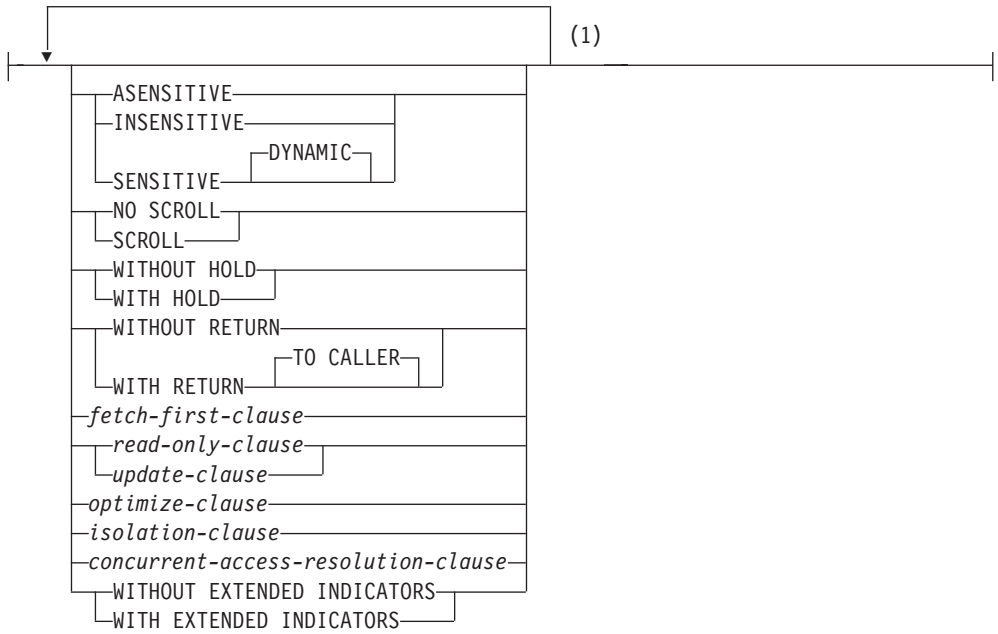
Authorization

The authorization rules are the same as those defined for the SQL statement specified by the PREPARE statement. For example, see Chapter 6, "Queries," on page 463 for the authorization rules that apply when a SELECT statement is prepared. The authorization ID is the run-time authorization ID.

Syntax



attribute-string:



Notes:

1 Each clause may be specified only 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.

concurrent-access-resolution-clause:

Description

statement-name
Names the prepared statement. If the name identifies an existing prepared statement, that prepared statement is destroyed. The name must not identify a prepared statement that is the SELECT statement of an open cursor.

INTO
If INTO is used, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQLDA specified by *descriptor-name*. Thus, the PREPARE statement:

```
EXEC SQL PREPARE S1 INTO :SQLDA FROM :V1;
```

is logically 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 Chapter 12, "SQLDA (SQL descriptor area)," on page 985. 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 Chapter 18, "Coding SQL statements in REXX applications," on page 1117.):

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 on techniques to determine the number of occurrences required, see "Determining how many occurrences of SQLVAR entries are needed" on page 987.

See "DESCRIBE" on page 769 for an explanation of the information that is placed in the SQLDA.

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.

The attributes are ignored if the prepared statement is not a *select-statement*.

attr-variable must identify a character-string or Unicode graphic-string 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*.

If *attr-variable* is a host variable, an indicator variable can be used to indicate whether or not 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 the sensitivity of the cursor to inserts, updates, or deletes that are made to the rows of the underlying the base tables. The sensitivity of the cursor determines whether DB2 can materialize the rows of the result into a temporary table of the cursor. The default is ASENSITIVE.

ASENSITIVE
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 "read-only-clause" on page 513.)

If SENSITIVE is specified, then a *fetch-first-clause* must not be specified.

INSENSITIVE
Specifies that once the cursor is opened, it does not have sensitivity to inserts, updates, or deletes performed by this or any other application process. 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 FOR UPDATE clause.

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.

NO SCROLL
Specifies that the cursor is not scrollable.

SCROLL
Specifies that the cursor is scrollable.

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 749.

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 749.

fetch-first-clause
Specifies that a maximum number of rows should be retrieved. For more information, see "fetch-first-clause" on page 497.

If a *fetch-first-clause* is specified, then an *update-clause* must not be specified.

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 513 and "update-clause" on page 512.

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 514.

isolation-clause
Specifies an isolation level at which the select statement is executed. For more information, see "isolation-clause" on page 515.

concurrent-access-resolution-clause
Specifies the type of concurrent access resolution to use for the select 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 is ignored if it is specified when the isolation level that is in effect is RR or UR.

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 USE AND KEEP EXCLUSIVE LOCKS clause and is ignored otherwise. Applicable scans include read-only scans.

WAIT FOR OUTCOME
Specifies that the database manager should 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 RR or UR is in effect.

WITHOUT EXTENDED INDICATORS or WITH EXTENDED INDICATORS
Specifies whether the values provided for indicator variables during execution of an INSERT, MERGE, 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 is ignored unless the statement is an INSERT with a VALUES clause, MERGE statement, or an UPDATE statement.

G DB2 for LUW supports prepare attributes as an application server, but does not
G support the ATTRIBUTES clause in embedded SQL.

FROM
Introduces the statement string. The statement string is the value of the specified *variable*.

variable
Identifies the *variable* that contains the statement string. The *variable* must identify a variable that is described in the application program in
| accordance with the rules for declaring character string variables. A global
| variable must not be specified. The variable must not have a CLOB data type, and an indicator variable must not be specified. In COBOL, the variable must be a varying-length string variable. In C, the variable must not be a NUL-terminated string.

The statement string must be one of the following SQL statements:

ALTER	INSERT	select-statement
COMMENT	LOCK TABLE	SET CURRENT DECFLOAT ROUNDING MODE
COMMIT	MERGE	SET CURRENT DEGREE
CREATE	REFRESH TABLE	SET ENCRYPTION PASSWORD
DECLARE GLOBAL TEMPORARY TABLE	RELEASE SAVEPOINT	SET PATH
DELETE	RENAME	SET SCHEMA
DROP	REVOKE	UPDATE
FREE LOCATOR	ROLLBACK	TRUNCATE
GRANT	SAVEPOINT	

The statement string must not:
- Begin with EXEC SQL.
- End with END-EXEC or a semicolon.
- Include references to variables. Global variables are allowed.

Notes

Parameter Markers: Although a statement string cannot include references to host variables, SQL variables, or SQL parameters, it may include *parameter markers* or global variables. Parameter markers 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 828 and "EXECUTE" on page 784.

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 invocation of a CAST specification is a "promise" that the data type of the parameter at run time will be of the data type specified or some data type that is assignable 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 data type that can be converted to VARCHAR(12). For more information, refer to "Assignments and comparisons" on page 77.

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 only in selected locations where variables are supported. These locations and the resulting data type are found in Table 55. The locations are grouped in this table into expressions, predicates and functions to assist in determining applicability of an untyped parameter marker.

Table 55. Untyped parameter marker usage

Untyped Parameter Marker Location	Data Type
Expressions (including select list, CASE and VALUES)	
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. [130]
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.
Labeled duration within a datetime expression with a unit type other than SECONDS. (Note that the portion of a labelled duration that indicates the type of units cannot be a parameter marker.)	Error

Table 55. Untyped parameter marker usage (continued)

Untyped Parameter Marker Location	Data Type
Labeled duration within a datetime expression with a type unit of SECONDS. (Note that the portion of a labelled duration that indicates the type of units cannot be a parameter marker.)	Error
Any other operand of a datetime expression (for instance 'timecol + ?' or '? - datecol').	Error
Both operands of a CONCAT operator	Error
One operand of a CONCAT operator when the other operand is a non-CLOB character data type	VARCHAR(*n*) where *n* is product-specific.
One operand of a CONCAT operator, when the other operand is a non-DBCLOB graphic data type	VARGRAPHIC(*n*) where *n* is product-specific.
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	Error
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 91 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 91 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 single-row VALUES clause within an INSERT statement.	The data type of the column. If the column is defined as a distinct type, then it is the source data type of the distinct type.[130]
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 distinct type, then it is the source data type of the distinct type.[130]
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 distinct type, then it is the source data type of the distinct type.[130]
As a value on the right side of a SET clause of an UPDATE statement or the VALUES clause of in insert operation of a MERGE statement.	The data type of the column. If the column is defined as a distinct type, then it is the source data type of the distinct type. [130]

Table 55. Untyped parameter marker usage (continued)

Untyped Parameter Marker Location	Data Type
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 INTO clause of the VALUES INTO statement	The data type of the associated expression. [130]
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)
Predicates	
Both operands of a comparison operator	Error
One operand of a comparison operator where the other operand is other than an untyped parameter marker or a distinct type.	The data type of the other operand. [130]
One operand of a comparison operator where the other operand is a distinct type.	Error
All operands of a BETWEEN predicate	Error
Two operands of a BETWEEN predicate (either the first and second, or the first and third)	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 91 on all operands that are other than untyped parameter markers.
All operands of an IN predicate, for example, ? IN (?,?,?)	Error
The 1st operand of an IN predicate where the right side is a subselect, for example, ? IN (subselect).	Data type of the selected column
The 1st operand of an IN predicate where the right side is not a subselect, for example, ? IN (?,A,B) or for example, ? IN (A,?,B,?).	Result of applying the "Rules for result data types" on page 91 on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers.
Any or all operands of the IN list of the IN predicate, for example, A IN (?,B,?).	Result of applying the "Rules for result data types" on page 91 on all operands of the IN predicate (operands to the left and right of the IN predicate) that are other than untyped parameter markers. In DB2 for z/OS, the operand to the left of the IN predicate is not included.
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

Table 55. Untyped parameter marker usage (continued)

Untyped Parameter Marker Location	Data Type
All three operands of the LIKE predicate.	Match expression (operand 1) and pattern expression (operand 2) VARCHAR(*n*), where *n* if product-specific. The escape expression (operand 3) is VARCHAR(*n*) where *n* is 1 or 2 depending on the default CCSID.
The match expression of the LIKE predicate.	Either VARCHAR(*n*) or VARGRAPHIC(*n*) depending on the data type of the first operand that is not an untyped parameter marker, where *n* is product-specific.
The pattern expression of the LIKE predicate.	Either VARCHAR(*n*) or VARGRAPHIC(*n*) or BLOB(*n*) depending on the data type of the match expression, where *n* is product-specific. For information on using fixed-length variables for the value of the pattern see LIKE Predicate Notes.
The escape expression of the LIKE predicate.	Either VARCHAR(*n*) or VARGRAPHIC(1) or BLOB(1) depending on the data type of the match expression, where *n* is 1 or 2 depending on the default CCSID.
Operand of the NULL predicate.	Error
Functions	
All operands of BITAND, BITANDNOT, BITOR, BITXOR, BITNOT, COALESCE, or VALUE.	Error
Any operand of COALESCE or VALUE where at least one operand is other than an untyped parameter marker.	Result of applying the "Rules for result data types" on page 91 on all operands that are other than untyped parameter markers.
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).
Both arguments of LOCATE, POSITION, or POSSTR	The data type and length are product-specific.
One argument of LOCATE, POSITION, or POSSTR when the other argument is a character data type.	VARCHAR(*n*) where *n* is product-specific.
One argument of LOCATE, POSITION, or POSSTR when the other argument is a graphic data type.	VARGRAPHIC(*n*) where *n* is product-specific.
One argument of LOCATE, POSITION, or POSSTR when the other argument is a binary data type.	The data type and length are product-specific.
The operand of LOWER, UPPER, LCASE, and UCASE.	Either VARCHAR(*n*) or DBCLOB(*n*) where the data type and *n* is product-specific.
First argument of SUBSTR or SUBSTRING	Either VARCHAR(*n*) or DBCLOB(*n*) where the data type and *n* is product-specific.
Second and third arguments of SUBSTR and SUBSTRING	INTEGER

Table 55. Untyped parameter marker usage (continued)

Untyped Parameter Marker Location	Data Type
The first argument of TIMESTAMP or TIMESTAMP_ISO	Error
The second argument of TIMESTAMP	TIME
First operand of TIMESTAMP_FORMAT	Either VARCHAR(*n*) or VARGRPAHIC(*n*) where the data type and *n* is product-specific.
The second operand of TIMESTAMP_FORMAT.	Error
First operand of TRANSLATE	Error
Second and third operands of TRANSLATE	Either VARCHAR(*n*) or VARGRAPHIC(*n*) depending on the data type of the first operand, where *n* is product-specific.
Fourth operand of TRANSLATE	VARCHAR(1) if the first operand is a character type. VARGRAPHIC(1) if the first operand is a graphic type.
The first operand of VARCHAR_FORMAT.	TIMESTAMP(12)
The second operand of VARCHAR_FORMAT.	Error
Array index of an ARRAY	BIGINT
Unary minus	DECFLOAT(34)
All other operands of all other scalar functions.	Error
Operand of an aggregate function.	Error
User-defined Routines	
Argument of a function	The data type of the parameter, as defined when the function was created
Argument of a procedure	The data type of the parameter, as defined when the procedure was created

Error checking: When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is invalid, a prepared statement is not created and an error is returned.

A product-specific option may be used to 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
DESCRIBE INPOUT    None
DECLARE CURSOR     Must be SELECT when the cursor is opened
EXECUTE            Must not be SELECT
```

130. If the data type is DATE, TIME, or TIMESTAMP, then CHAR(n), where n is product-specific.

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.

Prepared statement persistence: All prepared statements are destroyed when:[131]

- A CONNECT (Type 1) statement is executed.
- A prepared statement is associated with a release-pending connection and a successful commit occurs.

G In DB2 for z/OS, all prepared statements are destroyed when the unit of work
G ends except:

G
- if the SELECT statement whose cursor is declared with the option WITH HOLD
G persists over the execution of a commit operation if the cursor is open when the
G commit operation is executed

G
- if SELECT, INSERT, UPDATE, and DELETE statements that are bound with
G KEEPDYNAMIC(YES).

Scope of a statement: The scope of *statement-name* is the source program in which it is defined. A prepared statement can only be referenced 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.

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.

Examples

Example 1: Prepare and execute a statement other than a 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.
```

131. Prepared statements may be cached and not actually destroyed. However, a cached statement can only be used if the same statement is prepared again.

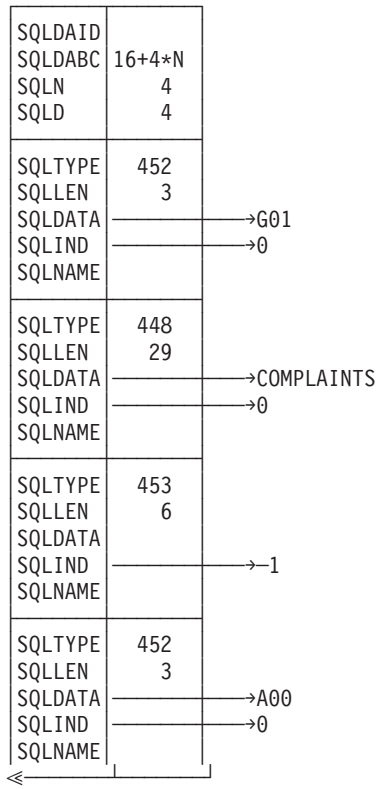
Example 2: Prepare and execute a non-select-statement as in example 1, except assume the statement to be prepared can contain any number of parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER  END-EXEC.

/* Set up the SQLDA */

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 issuing 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:

- Ownership of the table
- Database administrator authority.

Syntax

Description

table-name
Identifies the materialized query 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.

Examples

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

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

None required.

Syntax

Description

server-name **or** *variable*
Identifies a connection by the specified server name or the server name contained in the variable. 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 a global variable. In DB2 for z/OS, the maximum length of the value is 16. In DB2 for LUW, the maximum length of the value is 8.
- It must not be followed by an indicator variable
- If a reserved word is used as an identifier in SQL, it must be specified in uppercase and either as a delimited identifier or in a variable.
- 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 application process.

CURRENT
Identifies the current connection of the application process. The application process must be in the connected state.

An application server named CURRENT can only be identified by a variable or a delimited identifier.

ALL or ALL SQL
Identifies all existing connections of the application process (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 application process 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 application process 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 default application server 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 WITH HOLD.

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 TOROLAB;
```

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 them 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.

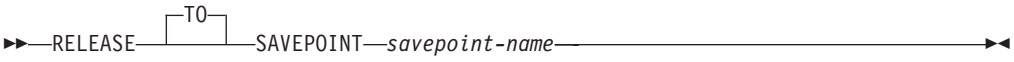
Invocation

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.

Examples

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 an existing table or index.

Invocation

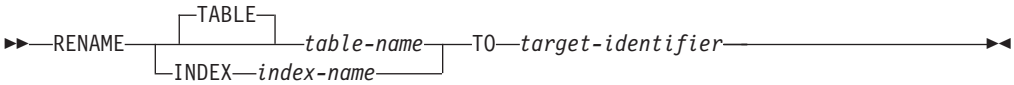
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the table or index
- Adminstrative authority.

Syntax



Description

TABLE *table-name*
Identifies the existing table that is to be renamed. The name, including the implicit or explicit qualifier, must identify a table that exists at the current server. It must not be the name of a catalog table, a materialized query table, or a declared temporary table.

The table must not be:

- referenced in any existing view definitions or materialized query table definitions
- referenced in triggered statements, and cannot have a trigger defined on it
- a parent or dependent table in any referential integrity constraints
- defined with any check constraints.

INDEX *index-name*
Identifies the existing index that is to be renamed. The name, including the implicit or explicit qualifier, must identify an index that exists at the current server. It must not be a system defined catalog index, or be defined on a declared temporary table.

target-identifier
Specifies the new name for the table or index without a qualifier. The qualifier of the *table-name* or *index-name* is used to qualify the new identifier for the table or index. The new qualified name must not identify a table, view, alias, or index that already exists at the current server.

Notes

Effects of the statement: The specified table 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: For a renamed table, any access plans that refer to that table are invalidated. For a renamed index, any access plans that refer to that index might be invalidated. See product documentation for details. For more information, see "Packages and access plans" on page 18.

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.

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
```

REVOKE (function or procedure privileges)

This form of the REVOKE statement revokes the privilege on a function or procedure.

Invocation

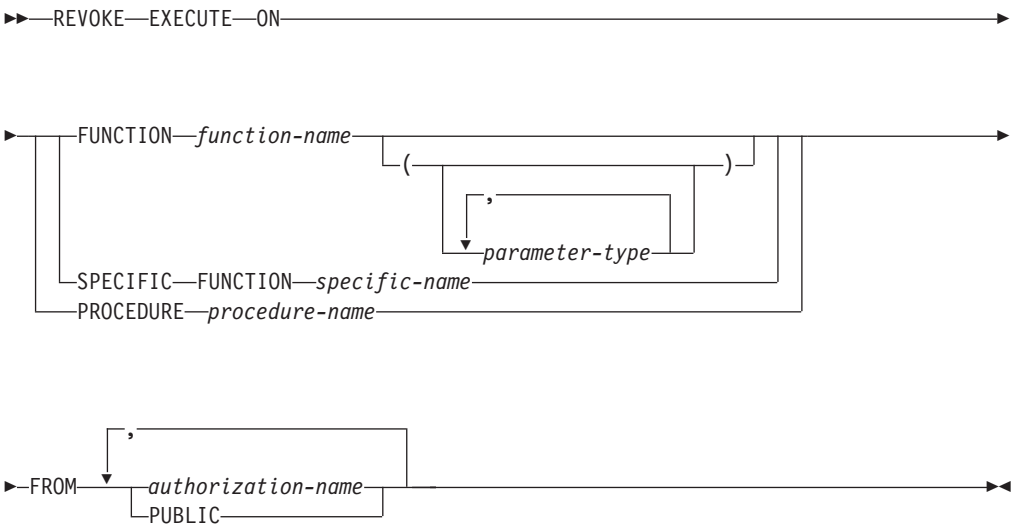
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

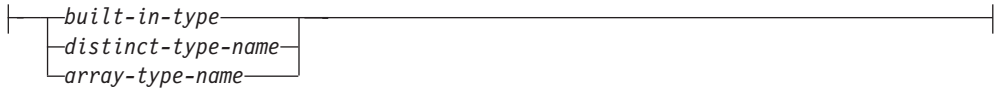
- Ownership of the function or procedure
- Security administrator authority.

Syntax

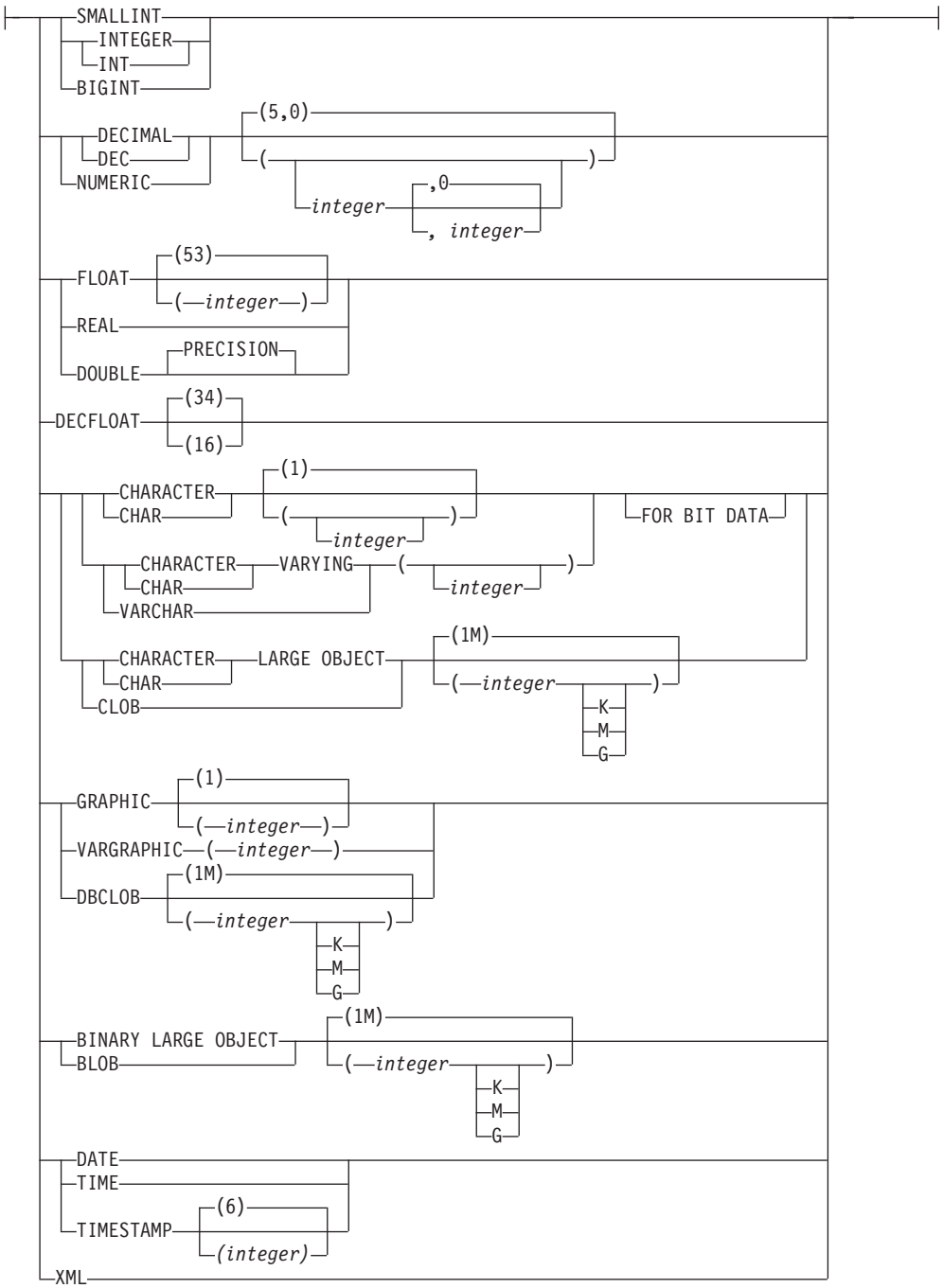


parameter-type:

data-type:



built-in-type:



Description

EXECUTE
Revokes the privilege to execute a function or procedure. The privilege may not be revoked if the authorization ID of the statement did not grant the EXECUTE privilege on the function or procedure. For more information see, "Authorization, privileges and object ownership" on page 21.

G A user with security administrator authority may revoke the EXECUTE privilege granted by others. The technique is product-specific.

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. The function can be identified by 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 from 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 user-defined type name is specified, the database manager searches the SQL path to resolve the schema name for the user-defined 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, DECIMAL() will be considered a match for a parameter of a function defined with a data type of DECIMAL(7,2). However, FLOAT cannot be specified with empty parentheses 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 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.

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.

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 *procedure-name*
Identifies the procedure from which the privilege is revoked. The *procedure-name* must identify a procedure that exists at the current server.

FROM
Identifies from whom the privilege is revoked.

authorization-name,...
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 21.

Notes

REVOKE restrictions: The EXECUTE privilege must not be revoked on a function or procedure if the *authorization-name* owns any of the following objects:

- A function that is sourced on the function
- A view that uses the function
- A trigger that uses the function or procedure
- A table that uses the function in a DEFAULT clause

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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 database administrator authority.

Examples

Revoke the EXECUTE privilege on procedure PROCA from PUBLIC.

```
REVOKE EXECUTE
  ON PROCEDURE PROCA
  FROM PUBLIC
```

REVOKE (package privileges)

This form of the REVOKE statement revokes the privilege to execute statements in a package.

Invocation

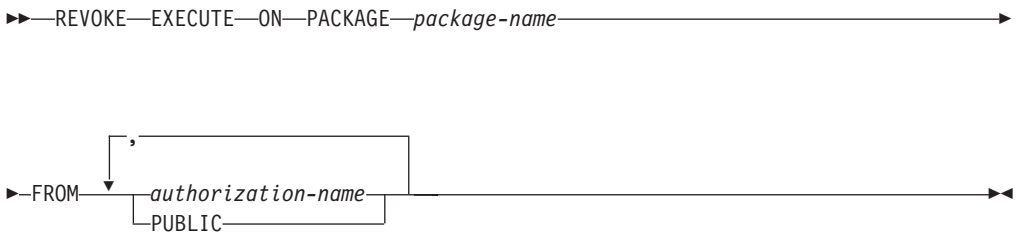
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the package
- Security administrator authority.

Syntax



Description

EXECUTE
Revokes the privilege to execute SQL statements in a package. The privilege may not be revoked if the authorization ID of the statement did not grant the EXECUTE privilege on the package. For more information see, "Authorization, privileges and object ownership" on page 21.

A user with security administrator authority may revoke the EXECUTE privilege granted by others. The technique is product-specific.

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 privilege is revoked.

authorization-name,...
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 21.

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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 database administrator authority.

Examples

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 revokes privilege on a sequence.

Invocation

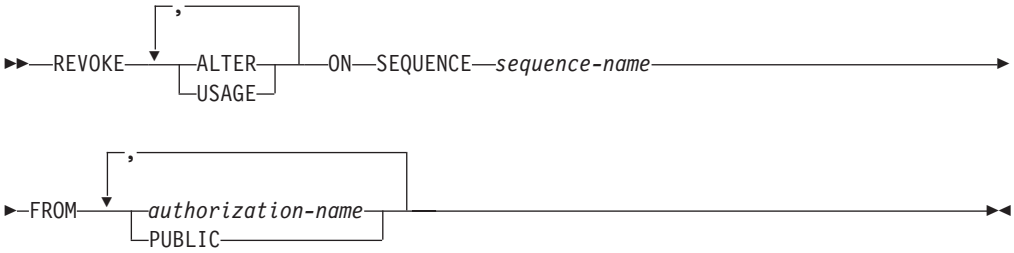
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the sequence
- Security administrator authority.

Syntax



Description

Each keyword revokes the privilege described, but only as it applies to the sequence named in the ON clause. The same keyword must not be specified more than once.

A privilege may not be revoked if the authorization ID of the statement did not grant the specified privilege on the sequence. For more information see, "Authorization, privileges and object ownership" on page 21.

A user with security administrator authority may revoke privileges granted by others. The technique is product-specific.

ALTER
Revokes the privilege to use the ALTER SEQUENCE statement on a sequence. The privilege may not be revoked if the authorization ID of the statement did not grant the ALTER privilege on the sequence. For more information see, "Authorization, privileges and object ownership" on page 21.

USAGE
Revokes the privilege to use the sequence in NEXT VALUE or PREVIOUS VALUE expressions. The privilege may not be revoked if the authorization ID of the statement did not grant the USAGE privilege on the sequence. For more information see, "Authorization, privileges and object ownership" on page 21.

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 privilege is revoked.

authorization-name,...
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 21.

Notes

REVOKE Restrictions: The USAGE privilege must not be revoked on a sequence if:

- The *authorization-name* owns a function or procedure that uses the sequence, or
- The *authorization-name* owns a table that has a trigger that uses the sequence.

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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 database administrator authority.

Examples

REVOKE the USAGE privilege from PUBLIC on a sequence called ORG_SEQ.

```
REVOKE USAGE
  ON SEQUENCE ORG_SEQ
  FROM PUBLIC
```

REVOKE (table and view privileges)

This form of the REVOKE statement revokes privileges on a table or view.

Invocation

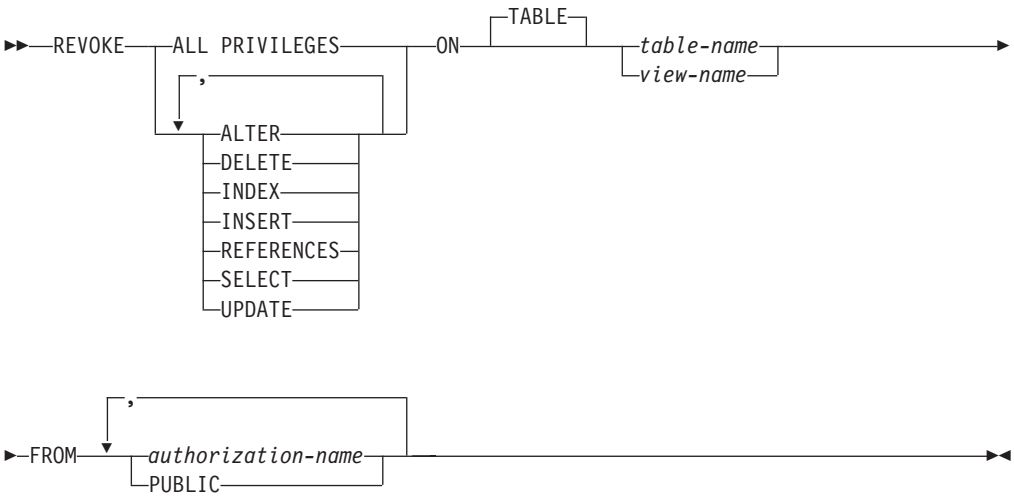
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the table or view
- Security administrator authority.

Syntax



Description

Each keyword revokes the privilege described, but only as it applies to the table or view named in the ON clause. The same keyword must not be specified more than once.

A privilege may not be revoked if the authorization ID of the statement did not grant the specified privilege on the table or view. For more information see, "Authorization, privileges and object ownership" on page 21.

A user with security administrator authority may revoke privileges granted by others. The technique is product-specific.

ALL PRIVILEGES
Revokes one or more privileges from each *authorization-name*. The privileges revoked are all those privileges on the identified table or view which were granted to the *authorization-name*.

ALTER
Revokes the privilege to alter the specified table or create a trigger on the specified table.

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 specified table is the parent.

SELECT
Revokes the privilege to create a view or read data from the specified table or view.

UPDATE
Revokes the privilege to update rows in the specified table or view.

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. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of privileges to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 21.

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

Revoking column privileges: The only way to revoke column privileges is to revoke the privilege from the entire table itself and then grant it again for each column.

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 database administrator authority.

Effect on views and access plans: Revoking a privilege that was used to create a view might cause the view to be dropped. Revoking a privilege that was used to create an access plan may invalidate the access plan. In both cases, the rules are product-specific.

Dependent privileges: In DB2 for z/OS, when a privilege is revoked from a user, every privilege dependent on that privilege is also revoked. For more information see the DB2 for z/OS product books. In DB2 for i and DB2 for LUW, privileges are not dependent upon other privileges.

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
```

REVOKE (type privileges)

This form of the REVOKE statement revokes the privilege on a user-defined type.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

G In DB2 for LUW, this statement is not supported. Instead, PUBLIC implicitly has
G the USAGE privilege on all user-defined types which cannot be revoked.

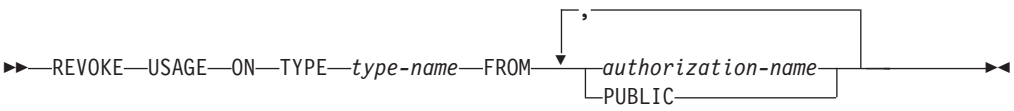
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the user-defined type
- | Security administrator authority.

Syntax

|



Description

USAGE
Revokes the privilege to use a user-defined type. The privilege may not be revoked if the authorization ID of the statement did not grant the USAGE privilege on the user-defined type. For more information see, "Authorization, privileges and object ownership" on page 21.

| A user with security administrator authority may revoke the USAGE privilege
G granted by others. The technique is product-specific.

| **ON TYPE** *type-name*
Identifies the user-defined type from which the privilege is revoked. The
| *type-name* must identify a user-defined type that exists at the current server.

FROM
Identifies from whom the privilege is revoked.

authorization-name,...
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 21.

Notes

REVOKE restrictions: The USAGE privilege must not be revoked on a user-defined type if:

- The *authorization-name* owns a function or procedure that uses the user-defined type, or
- The *authorization-name* owns a table that has a column that uses the user-defined type.

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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 database administrator authority.

Examples

Revoke the USAGE privilege on user-defined type SHOESIZE from user JONES.

```
REVOKE USAGE
  ON TYPE SHOESIZE
  FROM JONES
```

REVOKE (variable privileges)

This form of the REVOKE statement revokes privilege on a variable.

Invocation

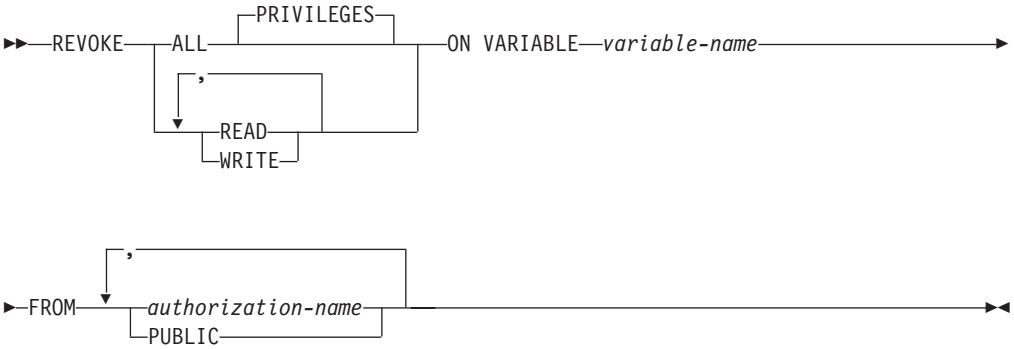
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the variable
- Security administrator authority.

Syntax



Description

Each keyword revokes the privilege described, but only as it applies to the variable named in the ON clause. The same keyword must not be specified more than once.

A privilege may not be revoked if the authorization ID of the statement did not grant the specified privilege on the variable. For more information see, "Authorization, privileges and object ownership" on page 21.

A user with security administrator authority may revoke privileges granted by others. The technique is product-specific.

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 variable that were granted to the *authorization-names*.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

READ
Revokes the privilege to read the value of the specified global variable. The privilege may not be revoked if the authorization ID of the statement did not grant the READ or WRITE privilege on the variable. For more information see, "Authorization, privileges and object ownership" on page 21.

WRITE
Revokes the privilege to assign a value to the specified global variable. The privilege may not be revoked if the authorization ID of the statement did not grant the READ or WRITE privilege on the variable. For more information see, "Authorization, privileges and object ownership" on page 21.

ON variable *variable-name*
Identifies the variable from which the privilege is revoked. The *variable-name* must identify a variable that exists at the current server.

FROM
Identifies from whom the privilege is revoked.

authorization-name,...
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, privileges and object ownership" on page 21.

Notes

REVOKE Restrictions: The READ or WRITE privilege must not be revoked on a variable if:

- The *authorization-name* owns a function or procedure that uses the variable, or
- The *authorization-name* owns a view that uses the variable, or
- The *authorization-name* owns a table that has a trigger that uses the variable.

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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 database administrator authority.

Examples

REVOKE the WRITE privilege on a global variable MYSCHEMA.MYJOB_PRINTER from user ZUBIRI.

```
REVOKE WRITE
  ON VARIABLE MYSCHEMA.MYJOB_PRINTER
  FROM ZUBIRI
```

ROLLBACK

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 that were made after a savepoint was set within the unit of work at the current server without ending the unit of work. Rolling back to a savepoint enables selected changes to be undone.

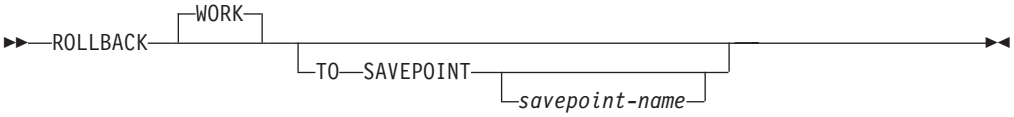
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

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 7, “Statements,” on page 519.

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.

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 occurs.

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 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 scope of 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 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 by the unit of work are released.
- For DB2 for z/OS, all statements that were prepared during the unit of work are destroyed. Any cursors associated with a prepared statement that is destroyed cannot be opened until the statement is prepared again.

ROLLBACK has no effect on the state of connections.

Other transaction environments: SQL ROLLBACK may not be available in other transaction environments, such as IMS and CICS. To do a rollback operation in these environments, SQL programs must use the call prescribed by their transaction manager.

Examples

Example 1: See “Examples” on page 597 under COMMIT which shows the use of 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 at the current server to identify a point in time to which relational database changes can be rolled back.

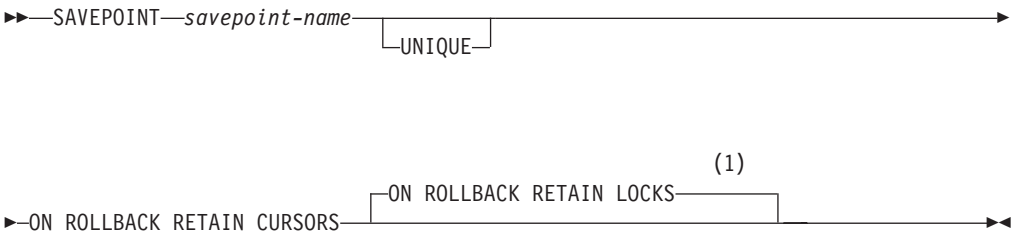
Invocation

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
Names 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.

Examples

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-statement" on page 505 and Chapter 6, "Queries," on page 463.

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

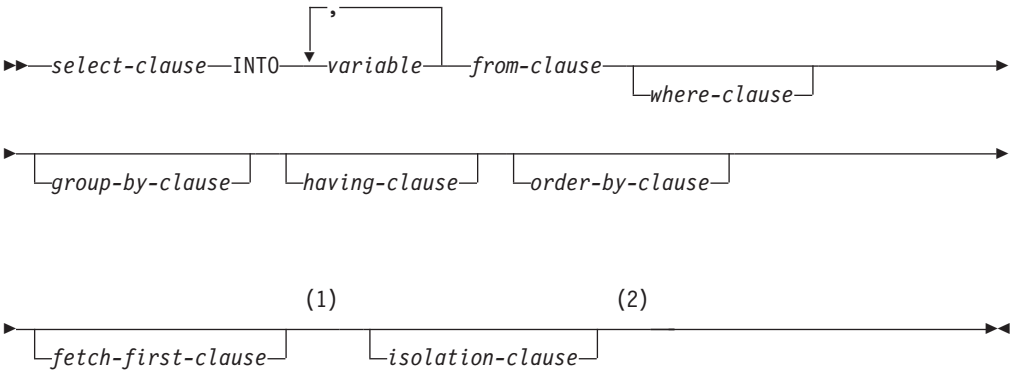
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the statement, one of the following:
  - The SELECT privilege on the table or view
  - Ownership of the table or view
- Database administrator authority.

If a global variable is referenced in the INTO clause, the privileges held by the authorization ID for the statement must include at least one of the following:

- The WRITE privilege on the global variable
- Database administrator authority

Syntax



Notes:

1 Only one row may be specified in the *fetch-first-clause*.

2 In DB2 for z/OS, if the *fetch-first-clause* and the *isolation-clause* are both specified, the *isolation-clause* must appear first.

Description

The result table is derived by evaluating the *isolation-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*, *fetch-first-clause*, and *select-clause*, in this order.

See Chapter 6, "Queries," on page 463 for a description of the *select-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*, *fetch-first-clause*, and *isolation-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 INTO, 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.

Notes

Variable assignment: Each assignment to a variable is performed according to the retrieval assignment rules described in "Assignments and comparisons" on page 77. 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 a value is null and the *variable* is a host variable, an indicator variable must be provided for that value.

If the specified variable is character and is not large enough to contain the result, '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 114.

If an assignment error occurs, the values in the variables are unpredictable.

Multiple variable 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). This error can be avoided by specifying FETCH FIRST ROW ONLY. 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, if the *variable* is a host variable, an indicator variable must be provided. The value of the variable is undefined. In this case, however, the indicator variable (if any) is set to the value of -2. Processing of the statement continues and a warning is returned. If an indicator variable is not providedand the *variable* is a host variable, 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. [132]

132. In DB2 for LUW, the database configuration parameter dft_sqlmathwarn must be set to yes for this behavior to be supported.

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 an error is returned. See "Datetime assignments" on page 83 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.

```
EXEC SQL  SELECT MAX(SALARY)
            INTO :MAX-SALARY
            FROM EMPLOYEE
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 (java.lang.String) and HOST_EDUCATE (java.lang.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 process by identifying one of its existing connections.

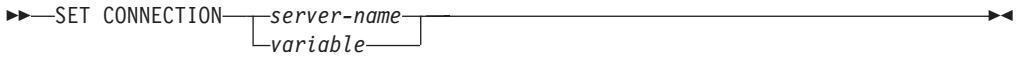
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

server-name **or** *variable*

Identifies the connection by the specified server name or the server name contained in the variable. If *variable* is specified:

- It must be a character-string variable with a length attribute that is not greater than 18. It must not be a global variable. In DB2 for z/OS, the maximum length of the value is 16. In DB2 for LUW, the maximum length of the value is 8.
- 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 application server S is placed in the SQLERRP field of the SQLCA. The format below applies if the application server is a DB2 product. The information has the form *pppvvrrm*, where:
  - *ppp* is:

    DSN for DB2 for z/OS

    QSQ for DB2 for i

SQL for DB2 for LUW
  - *vv* is a two-digit version identifier such as '09'.
  - *rr* is a two-digit release identifier such as '01'.
  - *m* is a one-digit modification level such as '0'.

  For example, if the server is Version 9 of DB2 for z/OS, the value would be 'DSN09010'.
- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. The contents are product-specific.
- Any previously current connection is placed in the dormant state.

If the SET CONNECTION statement is unsuccessful, the connection state of the application process 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 application process.

Examples

Execute SQL statements at TOROLAB, execute SQL statements at STLLAB, and then execute more SQL statements at TOROLAB.

```
EXEC SQL CONNECT TO TOROLAB;

  (execute statements referencing objects at TOROLAB)

EXEC SQL CONNECT TO STLLAB;

  (execute statements referencing objects at STLLAB)

EXEC SQL SET CONNECTION TOROLAB;

  (execute statements referencing objects at TOROLAB)
```

The first CONNECT statement creates the TOROLAB connection, the second CONNECT statement places it in the dormant state, and the SET CONNECTION statement returns it to the current state.

SET CURRENT DECFLOAT ROUNDING MODE

The SET CURRENT DECFLOAT ROUNDING MODE statement changes the value of the CURRENT DECFLOAT ROUNDING MODE special register.

G In DB2 for LUW, the value of the DECFLOAT rounding mode on a client can be
G confirmed to match that of the server by invoking the SET CURRENT DECFLOAT
G ROUNDING MODE statement. However, this statement cannot be used to change
G the rounding mode of the server.

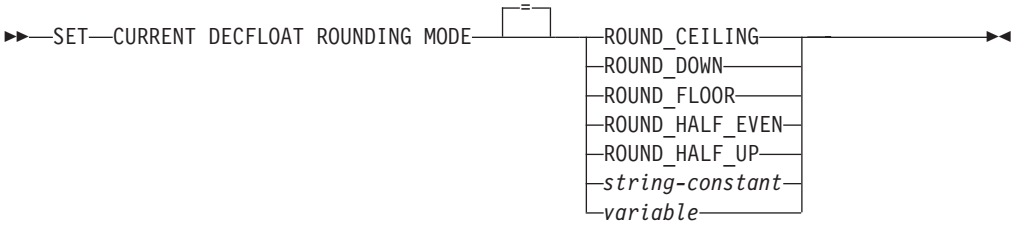
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

No authorization is required to execute this statement.

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_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.

string-constant
Specifies a character string constant. The content must be in uppercase characters.

The *string-constant* must have a length that does not exceed the maximum length of the CURRENT DECFLOAT ROUNDING MODE special register, after trailing blanks have been removed.

variable
Specifies a variable that contains the value for CURRENT DECFLOAT ROUNDING MODE.

The variable:

- Must be a CHAR or VARCHAR variable. It must not be a global variable. The actual length of the contents of the *variable* must not exceed the length of the CURRENT DECFLOAT ROUNDING MODE special register after trimming trailing blanks. See Chapter 9, "SQL limits," on page 959.
- Must not be the null value.
- Must have contents in uppercase characters. All characters are case-sensitive and are not converted to uppercase characters.

Notes

Transaction considerations: The SET CURRENT DECFLOAT ROUNDING MODE statement is not a commitable operation. ROLLBACK has no effect on the CURRENT DECFLOAT ROUNDING MODE.

Initial CURRENT DECFLOAT ROUNDING MODE: The initial value of CURRENT DECFLOAT ROUNDING MODE is product specific.

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 changes the value of 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.

Authorization

No authorization is required to execute this statement.

Syntax

Description

The value of CURRENT DEGREE is replaced by the string constant or variable.

string-constant
Specifies a character string constant. The content is not folded to uppercase.

The *string-constant* must have a length that does not exceed the maximum length of the CURRENT DEGREE special register.

variable
Specifies a variable that contains the value for CURRENT DEGREE. The content is not folded to uppercase.

The variable:

- Must be a CHAR or VARCHAR variable. It must not be a global variable. The actual length of the contents of the *variable* must not exceed the length of the CURRENT DEGREE special register after trimming trailing blanks. See Chapter 9, "SQL limits," on page 959.
- Must not be the null value.
- Must have contents in uppercase characters. All characters are case-sensitive and are not converted to uppercase characters.

The value of *string-constant* or *variable* must be one of:

1 No parallelism is allowed.

ANY
Specifies that the database manager can choose to use any degree of parallelism.

Notes

Transaction considerations: The SET CURRENT DEGREE statement is not a commitable operation. ROLLBACK has no effect on the CURRENT DEGREE.

Examples

Example 1: The following statement sets the CURRENT DEGREE special register to inhibit parallelism.

```
SET DEGREE = '1'
```

Example 2: The following statement sets the CURRENT DEGREE special register to allow parallelism.

```
SET DEGREE = 'ANY'
```

SET ENCRYPTION PASSWORD

The SET ENCRYPTION PASSWORD statement sets the default password 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" on page 267 and "DECRYPT_BIT and DECRYPT_CHAR" on page 260.

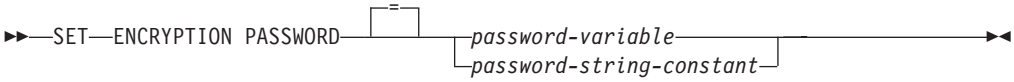
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

No authorization is required to execute this statement.

Syntax



Description

password-variable
Specifies a variable that contains an encryption password. The variable:

- Must be a CHAR or VARCHAR variable. It must not be a global 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 that contains an encryption password. 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. All characters are case-sensitive and are not converted to uppercase characters.

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.

Transaction considerations: The SET ENCRYPTION PASSWORD statement is not a commitable operation. ROLLBACK has no effect on the default encryption password.

Initial encryption password value: The initial value of the default encryption password is the empty string ('').

Encryption password scope: The scope of the default encryption password is the connection.

Examples

Set the ENCRYPTION PASSWORD to the value in :hv1.

```
SET ENCRYPTION PASSWORD :hv1
```

SET PATH

The SET PATH statement changes the value of the CURRENT PATH special register.

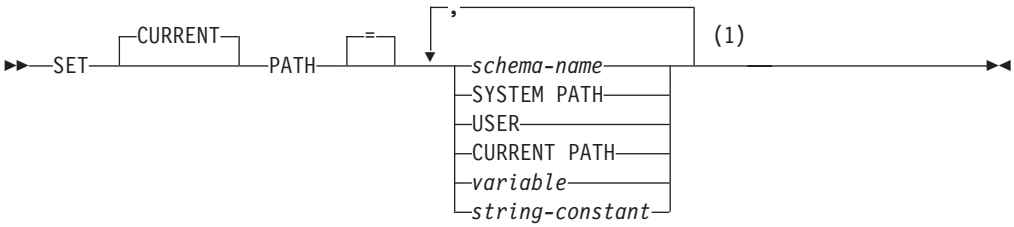
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

No authorization is required to execute this statement.

Syntax



Notes:

1 SYSTEM PATH, USER, and CURRENT PATH may each be specified at most once on the right side of the statement.

Description

schema-name
Identifies a schema. No validation that the schema exists is made at the time the SQL path is set. For example, if a *schema-name* is misspelled, it could affect the way subsequent SQL operates. Although not recommended, PATH can be specified as a *schema-name* if it is specified as a delimited identifier.

SYSTEM PATH
Specifies the schema names for the system path for the platform.

USER
Specifies the value of the USER special register.

CURRENT PATH
Specifies the value of the CURRENT PATH special register before the execution of this statement.

variable
Specifies a variable that contains a schema name.

The variable:

- Must be a CHAR or VARCHAR variable. It must not be a global variable. The actual length of the contents of the *variable* must not exceed the length of a schema name. See Chapter 9, "SQL limits," on page 959.
- Must not be followed by an indicator variable.
- Must not be the null value.

- Must include a schema name that is left justified and conforms to the rules for forming an ordinary identifier. The schema name must not be specified as a delimited identifier.
- 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.
- Must not contain SYSTEM PATH, USER, or CURRENT PATH.

string-constant
Specifies a string contant that contains a schema name. The string constant:

- Must have a length that does not exceed the maximum length of a schema name. See Chapter 9, “SQL limits,” on page 959.
- Must include a schema name that is left justified and conforms to the rules for forming an ordinary identifier. The schema name must not be specified as a delimited identifier.
- Must not contain lowercase letters or characters that cannot be specified in an ordinary identifier.
- Must not contain SYSTEM PATH, USER, or CURRENT PATH.

Notes

Transaction considerations: The SET PATH statement is not a commitable operation. ROLLBACK has no effect on the SQL path.

Rules for the content of the SQL path:

- A schema name cannot appear more than once in the SQL 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, changing each double quote character to two double quote characters within the schema name as necessary, and then separating each schema name by a comma. The length of the resulting string cannot exceed 254. See Chapter 9, “SQL limits,” on page 959 for more information.
- A schema name that does not conform to the rules for an ordinary identifier (for example: a schema name that contains lowercase characters or characters that cannot be specified in an ordinary identifier), must be specified as a *delimited schema name* and must not be specified within a variable or string constant. If the schema name changes dynamically in the application then the SET PATH statement must be dynamically prepared using the PREPARE and EXECUTE statements rather than changing the content of a variable.
- 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 on DB2 for z/OS, assuming that the current value of the USER special register is SMITH, then SET PATH = SYSIBM, SYSPROC, USER, "USER" results in a CURRENT PATH value of "SYSIBM","SYSPROC","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.

G **The system path:** SYSTEM PATH refers to the system path for a platform. The
G schemas in the *system path* are platform-specific.

G • For DB2 for LUW and DB2 for z/OS, the schemas of the system path are
G "SYSIBM", "SYSFUN", and "SYSPROC", and SYSTEM PATH is the same as
G specifying "SYSIBM", "SYSFUN", "SYSPROC".

G • For DB2 for i, the schemas of the system path are "QSYS" and "QSYS2", and
G SYSTEM PATH is the same as specifying "QSYS", "QSYS2".

When using the SET PATH statement, the system path must be specified explicity using SYSTEM PATH or implicitly by using CURRENT PATH which already includes the system path.

Using the SQL path: The CURRENT PATH special register specifies the SQL path used to resolve user-defined distinct types, functions and procedures in dynamic SQL statements. See “SQL path” on page 52.

Examples

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 SCHEMA

The SET SCHEMA statement changes the value of the CURRENT SCHEMA special register.

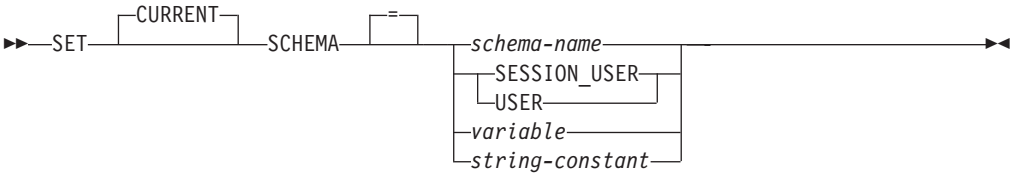
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

No authorization is required to execute this statement.

Syntax



Description

schema-name
Identifies a schema. No validation that the schema exists is made at the time that 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
Specifies the value of the SESSION_USER special register.

variable
Specifies a variable that contains a schema name. The content is not folded to uppercase.

The variable:

- Must be a CHAR or VARCHAR variable. It must not be a global variable. The actual length of the contents of the *variable* must not exceed the length of a schema name. See Chapter 9, "SQL limits," on page 959.
- Must not be followed by an indicator variable.
- Must not be the null value.
- Must include a schema name that is left justified and conforms to the rules for forming an ordinary or delimited identifier.
- Must be padded on the right with blanks if the variable is fixed length character.
- Must not contain SESSION_USER or USER.

string-constant
Specifies a string constant that contains a schema name. The content is not folded to uppercase.

The *string-constant*:

- Must have a length that does not exceed the maximum length of a schema name. See Chapter 9, “SQL limits,” on page 959.
- Must include a schema name that is left justified and conforms to the rules for forming an ordinary or delimited identifier.
- Must not contain SESSION_USER or USER.

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 commitable 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.

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 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.

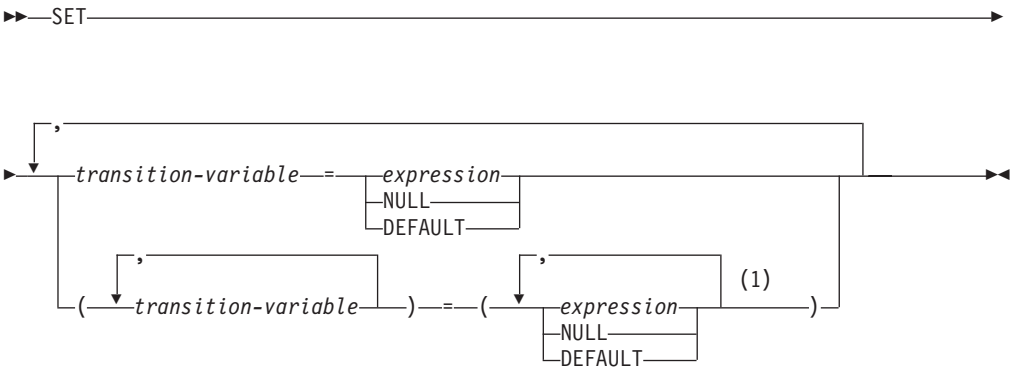
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The UPDATE privilege on any column associated with a *transition-variable* that occurs on the left side of the SET transition-variable statement in a before trigger
- Database administrator authority.

The privileges required to set a *transition-variable* is checked at the time the trigger is created. For more information, see "CREATE TRIGGER" on page 718.

Syntax



Notes:

1 The number of *expression*s, NULLs, and DEFAULTs must match the number of *transition-variables*.

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-variables* according to the storage assignment rules. For more information see "Assignments and comparisons" on page 77. DB2 for LUW uses retrieval assignment rules.

expression
Indicates the new value of the *transition-variable*. The expression is any expression of the type described in "Expressions" on page 130 that does not include an aggregate function, except when it occurs within a *scalar-fullselect*. If the expression contains a *scalar-fullselect*, the *scalar-fullselect* cannot reference columns of the triggering table.

An *expression* may contain references to OLD and NEW *transition-variables*. If the CREATE TRIGGER statement contains both OLD and NEW clauses, references to *transition-variables* must be qualified by the appropriate *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.

Notes

Multiple assignments: If more than one assignment is included in the same SET clause, all *expressions* are evaluated before the assignments are performed. Thus, references to *transition-variables* in an expression are always the value of the *transition-variable* prior to any assignment in the single 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 assigns values to variables.

Invocation

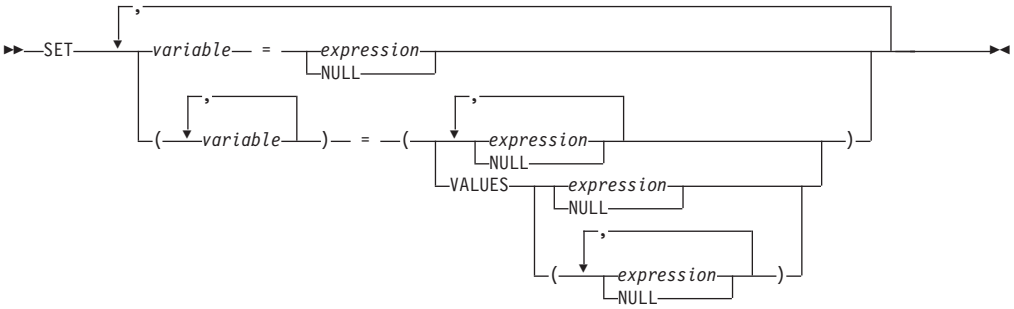
This statement can be embedded in an application program. It is an executable statement that can by dynamically prepared if the targets of all assignments are global variables. It must not be specified in REXX.

Authorization

If a global variable is specified on the left hand side of the assignment, the privileges held by the authorization ID for the statement must include at least one of the following:

- The WRITE privilege on the global variable
- Database administrator authority

Syntax



Description

variable, ...
Identifies one or more variables that must be declared in accordance with the rules for declaring variables.

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 *variable* and then all the values, for example, (*variable*, *variable*) = (*expression*, *expression*).

The data type of each variable must be compatible with its corresponding *expression*. Each assignment is made according to the rules described in "Assignments and comparisons" on page 77. 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 and the *variable* is a host variable, 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.

expression
Specifies the new value of the variable. The *expression* is any expression of the type described in "Expressions" on page 130. It must not include a column name.

The CURRENT SERVER special register can only be referenced in a SET variable statement that results in the assignment of a single variable and not those that result in setting more than one value.

NULL
Specifies that the new value for the variable is the null value.

VALUES
Specifies the values to be assigned to the corresponding variables. When more than one value is specified, the values must be enclosed in parentheses. The *expression* values are assigned to each corresponding *variable*.

The CURRENT SERVER special register can only be referenced in a VALUES clause that results in the assignment of a single variable and not those that result in setting more than one value.

Notes

Variable assignment: Each assignment to a variable is performed according to the retrieval assignment rules described in "Assignments and comparisons" on page 77. If a value is null and the *variable* is a host variable, an indicator variable must be provided for that value.

If the specified variable is character and is not large enough to contain the result, '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 114.

If an assignment error occurs, the values in the variables are unpredictable.

Multiple assignments: If more than one assignment is included in the same SET statement, all *expression*s are completely evaluated before the assignments are performed. Thus, references to a target variable in an *expression* 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);
```

TRUNCATE

The TRUNCATE statement deletes all of the rows from 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:

- The DELETE privilege on the table
- Ownership of the table.
- Database administrator authority

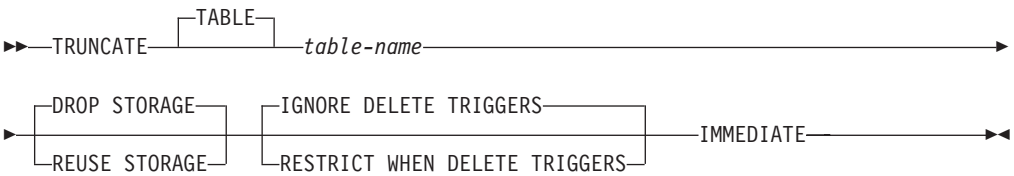
If the IGNORE DELETE TRIGGERS option is specified, the privileges held by the authorization ID of the statement must includeat least one of the following:

- The ALTER privilege on the table.
- Ownership of the table.
- Database administrator authority

If row access control or column access control is activated for the table, the privileges held by the authorization ID of the statement must include the following:

- Ownership of the table.
- Database administrator authority

Syntax



Description

table-name
Identifies the table from which rows are to be deleted. The name must identify a table that exists at the current server. It must not identify a catalog table or a view.

DROP STORAGE or REUSE STORAGE
Specifies whether to drop or reuse the existing storage that is allocated for the table.

DROP STORAGE
All storage allocated for the table is released and made available. This is the default.

REUSE STORAGE
All storage allocated for the table will continue to be allocated for the table, but the storage will be considered empty.

IGNORE DELETE TRIGGERS or RESTRICT WHEN DELETE TRIGGERS
Specifies what to do when delete triggers are defined on the table.

IGNORE DELETE TRIGGERS
Specifies that any delete triggers that are defined for the table are not activated by the truncation operation. This is the default.

RESTRICT WHEN DELETE TRIGGERS
Specifies that an error is returned if delete triggers are defined on the table

IMMEDIATE
Specifies that the truncate operation is processed immediately and cannot be undone. The statement must be the first statement in a transaction.

The truncated table is immediately available for use in the same unit of work. Although a ROLLBACK statement is allowed to execute after a TRUNCATE statement, the truncate operation is not undone, and the table remains in a truncated state. For example, if another data change operation is done on the table after the TRUNCATE IMMEDIATE statement and then the ROLLBACK statement is executed, the truncate operation will not be undone, but all other data change operations are undone.

The truncate operation cannot be performed if any session has a cursor open on the table or holds a lock on the table.

Notes

Referential Integrity: The identified table cannot be a parent table in a referential constraint.

Number of rows deleted: If no rows exist in the table, a SQLSTATE value of '02000' is returned.

Examples

Example 1: Empty an unused inventory table regardless of any existing triggers and return its allocated space.

```
TRUNCATE TABLE INVENTORY
     DROP STORAGE
     IGNORE DELETE TRIGGERS
     IMMEDIATE;
```

Example 2: Empty an unused inventory table regardless of any existing triggers but preserve its allocated space for later reuse.

```
TRUNCATE TABLE INVENTORY
     REUSE STORAGE
     IGNORE DELETE TRIGGERS
     IMMEDIATE;
```

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 for 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 can 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:

- The UPDATE privilege for the table or view
- The UPDATE privilege on each column to be updated
- Ownership of the table [133]
- Database administrator authority.

If the right side of *assignment-clause* contains a reference to a column of the table or view, or if *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 for the table or view [134]
- Ownership of the table or view
- Database administrator authority.

If the statement includes a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following:

- For every table or view identified in the subquery:
  - The SELECT privilege on the table or view, or
  - Ownership of the table or view
- Database administrator authority.

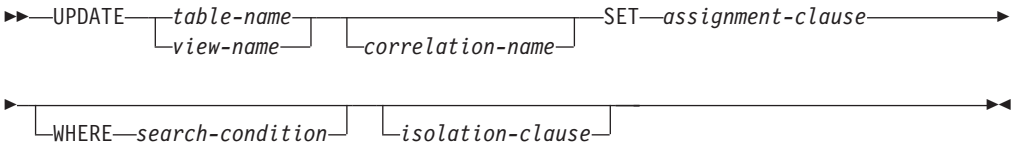
133. The UPDATE privilege on a view is only inherent in database administrator authority. Ownership of a view does not necessarily include the UPDATE privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.

134. In DB2 for z/OS, and DB2 for LUW, the authorization ID of the statement only requires the UPDATE privilege for the table or view. To require the SELECT privilege, a standards option must be in effect. For DB2 for z/OS use the program preparation option SQLRULES(STD) or set the CURRENT RULES special register to 'STD'. For DB2 for LUW, use the program preparation option LANGLEVEL SQL92E.

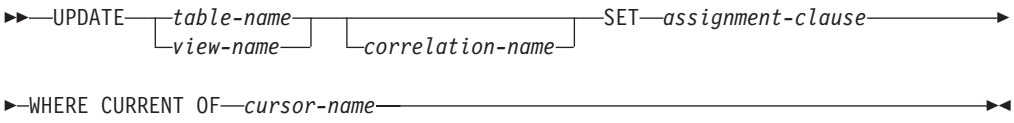
For more information about the subquery authorization rules, see Chapter 6, "Queries," on page 463.

Syntax

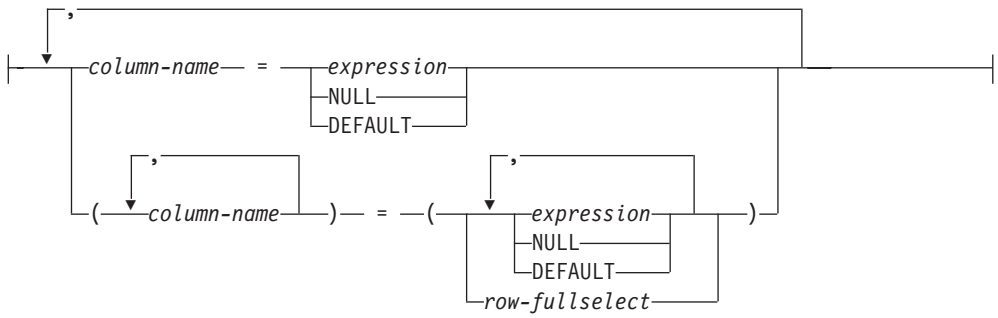
Searched UPDATE:



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 view that is not updatable. For an explanation of updatable views, see "CREATE VIEW" on page 743.

correlation-name
Can be used within *search-condition* or *assignment-clause* to designate the table or view. For an explanation of *correlation-name*, see "Correlation names" on page 108.

SET
Introduces the assignment of values to column names.

assignment-clause

If *row-fullselect* is specified, the number of columns in the result of *row-fullselect* must match the number of *column-names* that are specified. If *row-fullselect* is not specified, the number of *expressions*, and **NULL** and **DEFAULT** keywords must match the number of *column-names* that are specified.

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 be an updatable column. The same column name must not be specified more than once.

For a Positioned UPDATE:

- If the FOR UPDATE clause was specified in the *select-statement* of the cursor, each *column-name* must also appear in the FOR UPDATE clause.
- If the FOR UPDATE clause was not specified in the *select-statement* of the cursor, the name of any updatable column may be specified.[135]

For more information, see "update-clause" on page 512.

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.

expression

Indicates the new value of the column. The *expression* is any expression of the type described in "Expressions" on page 130, that does not include an aggregate function.

A *column-name* in an expression must name a column of the named table or view. For each updated row, the value of the column in the expression is the value of the column in the row before the row is updated.

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.

NULL

Specifies the null value as the new value of the column. Specify NULL only for nullable columns.

DEFAULT

DEFAULT Specifies that the default value is assigned to a column. The value that is assigned depends on how the column is defined, as follows:

- If the column is defined using the IDENTITY clause, the column is generated by the database manager.
- If the column is defined using the WITH DEFAULT clause, the value is set to the default that is defined for the column.
- If the column is defined without specifying the WITH DEFAULT clause, the IDENTITY clause, or the NOT NULL clause, the value is NULL.
- If the column was defined using the ROW CHANGE TIMESTAMP clause, the value is generated by the database manager.

135. In DB2 for z/OS, and DB2 for LUW, a program preparation option must be used if the UPDATE clause is not specified and the cursor is referenced in subsequent Positioned UPDATE statements. If this program preparation option is not used and the UPDATE clause is not specified, the cursor cannot be referenced in a Positioned UPDATE statement. For DB2 for z/OS the program preparation option is STDSQL(YES) or NOFOR, for DB2 for LUW it is LANGLEVEL SQL92E.

If the NOT NULL clause is used and neither the WITH DEFAULT clause nor the GENERATED clause is used, the DEFAULT keyword cannot be specified for that column.

DEFAULT must be specified for an identity column defined as GENERATED ALWAYS.

For information on default values of data types, see the description of the DEFAULT clause for CREATE TABLE in "CREATE TABLE" on page 688.

row-fullselect
Specifies a fullselect that returns a single row. The result column values are assigned to each corresponding *column-name*. If the fullselect returns no rows, the null value is assigned to each column; an error occurs if any column to be updated is not nullable. An error also occurs if there is more than one row in the result.

For a positioned update, if the table or view that is the object of the UPDATE statement is used in the fullselect, a column from the instance of the table or view in the fullselect must not be the same as *column-name*, a column being updated.

A *row-fullselect* may contain references to columns of the target table of the UPDATE statement. For each row that is updated, the value of such a column in an 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
Specifies a search condition, as described in "Search conditions" on page 178. Each *column-name* in the *search-condition*, other than in a subquery, must name a column of the table or view. The *search-condition* must not include a subquery where the base object of both the UPDATE and the subquery is the same table.

The *search-condition* is applied to each row of the table or view and the updated rows are those for which the result of the *search-condition* is true.

If *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.

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 749.

The table or view specified must also be identified in the FROM clause of the *select-statement* of the cursor, and the cursor must be updatable. For an explanation of updatable cursors, see "DECLARE CURSOR" on page 749.

When the UPDATE statement is executed, the cursor must be open and positioned on a row and that row is updated.

G In DB2 for z/OS, if a Positioned UPDATE statement is embedded in a
G program, the associated DECLARE CURSOR statement must include a
G *select-statement* rather than a *statement-name*.

This form of UPDATE must not be used with a cursor that references a view on which an INSTEAD OF trigger is defined, even if the view is an updatable view.

isolation-clause
Specifies the isolation level used by the statement.

WITH

Introduces the isolation level, which may be one of:
- *RR* Repeatable read
- *RS* Read stability
- *CS* Cursor stability

If *isolation-clause* is not specified, the default isolation is used. See "Isolation level" on page 28 for a description of how the default is determined.

UPDATE Rules

Assignment: Update values are assigned to columns in accordance with the storage assignment rules described in "Assignments and comparisons" on page 77.

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-subselect* 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 743.

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 return 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.

Updating rows in a table for which row or column access control is active: When an UPDATE statement is issued for a table for which row or column access control is active, the rules specified in the enabled row permissions or column masks determine whether the row can be updated. Typically those rules are based on the authorization ID of the statement. The following describes how enabled row permissions and column masks are used during UPDATE:

- Row permissions are used to identify the set of rows to be updated.

  When multiple enabled row permissions are defined for a table, a row access control search condition is derived by application of the logical OR operator to the search condition in each enabled permission. This row access control search condition is applied to the table to determine which rows are accessible to the authorization ID or role of the UPDATE statement. If the WHERE clause is specified in the UPDATE statement, the user-specified predicates are applied on the accessible rows to determine the rows to be updated. If there is no WHERE clause, the accessible rows are the rows to be updated.

  Column masks are not applicable in this step.

  If the table is not enforced by row access control, the WHERE clause determines the rows to be updated, otherwise all rows in the table are to be updated.
- If there are rows to be updated, the following rules determine whether those rows can be updated:
  - For every column to be updated, the new value of the column must not be affected by enabled column masks whose columns are referenced when deriving the new value.

    When a column is referenced while deriving the values of a new row, if the column has an enabled column mask, the masked value is used to derive the new values. If the object table is also column access control activated, the column mask applied to derive the new values must ensure the evaluation of the access control rules defined in the column mask resolves the column to itself, not to a constant or an expression. If the column mask does not mask the column to itself, the new value cannot be used for update and an error is returned at run time.
  - If the rows are updatable, and there is a BEFORE UPDATE trigger for the table, the trigger is activated.

    Within the trigger actions, the new values for update might be modified in transition variables. When the final values are returned from the trigger, the new values are used for the update.
  - The rows that are to be updated must conform to the enabled row permissions:

    For each row that is to be updated, the old values are replaced with the new values that were specified in the UPDATE statement. A row that conforms to the enabled row permissions is a row that, if updated, can be retrieved using the derived row access control search condition.
  - If the rows are updatable, and there is an AFTER UPDATE trigger for the table, the trigger is activated.

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 variable: 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, changes from this statement, referential constraints, and any triggered SQL statements are rolled back.

It is possible for an error to occur that makes the state of the cursor unpredictable.

Number of rows updated: When an UPDATE statement is completed, SQLERRD(3) in the SQLCA shows the number of rows that qualified for the update operation. In the context of an SQL procedure statement, the value can be retrieved using the ROW_COUNT variable of the GET DIAGNOSTICS statement. For a description of the SQLCA, see Chapter 11, “SQLCA (SQL communication area),” on page 981.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful UPDATE statement. Until these locks are released by a commit or rollback operation, an updated row can only be accessed by:

- the application process that performed the update,
- another application process using isolation level UR through a read-only cursor, a SELECT INTO statement, or a subquery.

The locks can prevent other application processes from performing operations on the table.

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.

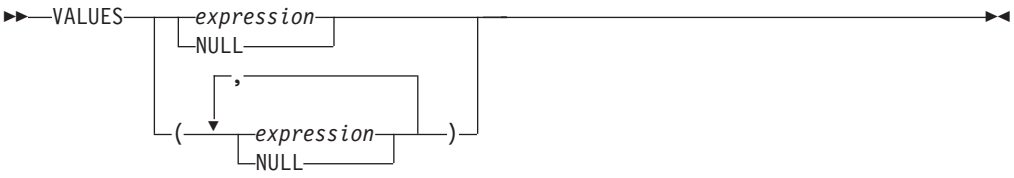
Invocation

This statement can only be used in the triggered action of a CREATE TRIGGER statement.

Authorization

None required.

Syntax



Description

VALUES
Introduces a single row consisting of one of more columns.

expression
Specifies any expression of the type described in "Expressions" on page 130.

NULL
Specifies the null value.

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.

Examples

Create an after trigger EMPISRT1 that invokes user-defined function NEWEMP when the trigger is activated. An insert operation on table EMPLOYEE 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

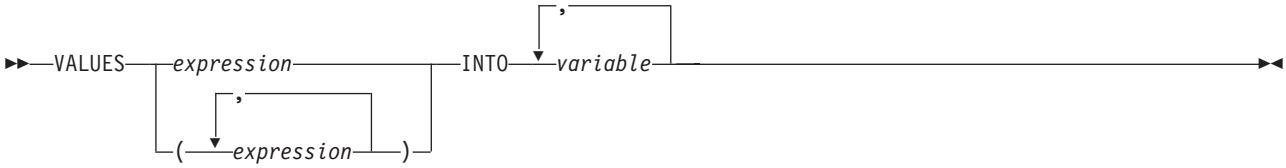
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

If a global variable is referenced in the INTO clause, the privileges held by the authorization ID for the statement must include at least one of the following:

- The WRITE privilege on the global variable
- Database administrator authority

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 130. The expression must not include a column name. Host structures are not supported.

INTO *host 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 77. Assignments are made in sequence through the list. 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'. See Chapter 11, "SQLCA (SQL communication area)," on page 981. Note that there is no warning if there are more variables than values. If a value is null and the *variable* is a host variable, an indicator variable must be provided for that value.

If the specified variable is character and is not large enough to contain the result, '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 114.

If an assignment error occurs, the values in the variables are unpredictable.

Multiple variable 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 list of expressions are always the value of the variable prior to any assignment in the VALUES INTO statement.

Expression evaluation considerations: If an error occurs while evaluating a *expression* 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, if the *variable* is a host variable, an indicator variable must be provided. The value of the variable is undefined. In this case, however, the indicator variable (if any) is set to the value of -2. Processing of the statement continues and a warning is returned. If an indicator variable is not provided and the *variable* is a host variable, 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.[136]

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 an error is returned. See “Datetime assignments” on page 83 for details.

Special register considerations: The special register CURRENT SERVER can be referenced only in a VALUES INTO statement that results in the assignment of a single variable and not those that result in setting more than one value.

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;
```

136. In DB2 for LUW, the database configuration parameter dft_sqlmathwarn must be set to yes for this behavior to be supported.

WHENEVER

The WHENEVER statement specifies the action to be taken when a specified exception condition occurs.

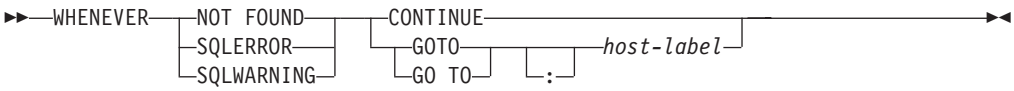
Invocation

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 "Handling SQL errors and warnings in Java" on page 1111 or "Handling SQL errors and warnings in REXX" on page 1121 for more information.

Authorization

None required.

Syntax



Description

The NOT FOUND, SQLERROR, or SQLWARNING clause is used to identify the type of exception condition. See Chapter 13, "SQLSTATE values—common return codes," on page 997.

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 GOTO clause is used to specify the next statement to be executed when the identified type of exception condition exists.

CONTINUE
Specifies the next sequential statement 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

(NOT FOUND, SQLERROR, and SQLWARNING). 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.

Subroutines are supported in COBOL and C. However, normal COBOL and C scoping rules are not followed. That is, the last WHENEVER statement specified in the program source prior to the subroutine remains in effect for that subroutine. The label referenced in the WHENEVER statement must be duplicated within that subroutine. Alternatively, the subroutine could specify a new WHENEVER statement.

Examples

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.
```

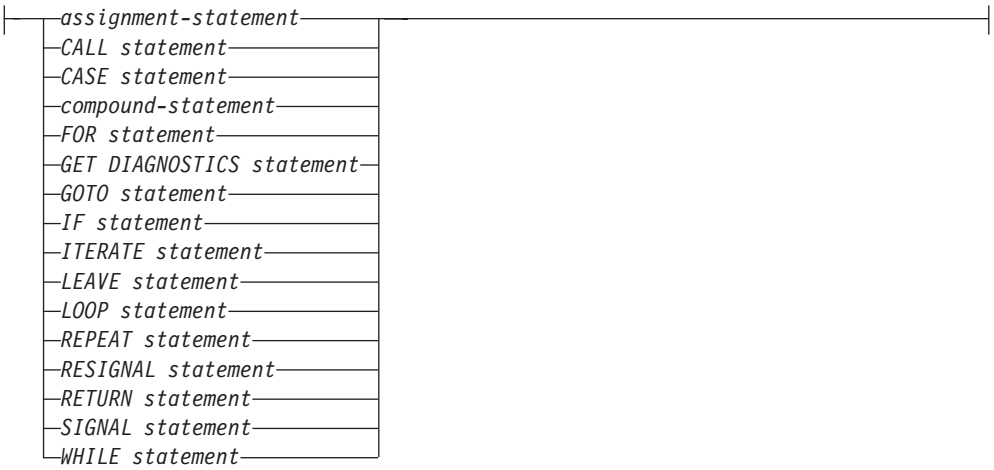
Example 3: Go to the label ENDDATA for any statement that does not return data when expected to do so.

```
EXEC SQL  WHENEVER NOT FOUND GOTO ENDDATA  END-EXEC.
```

Chapter 8. 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 procedures are created by specifying LANGUAGE SQL and an SQL routine body on the CREATE PROCEDURE statement. The SQL routine body must be a single SQL statement which may be an SQL control statement. Authorization requirements for control statements depend on the type of statement and objects referenced in the statement. For instance, a GOTO statement requires no authority but a CALL statement requires at least EXECUTE privilege on the procedure.

The remainder of this chapter contains a description of the control statements including syntax diagrams, semantic descriptions, usage notes, and examples of the use of the statements that constitute the SQL routine body. There is also a section on referencing SQL parameters and variables found in "References to SQL parameters and SQL variables" on page 909. There are two common elements that are used in describing specific SQL control statements. These are:

- SQL control statements as described above
- "SQL-procedure-statement" on page 916.

For syntax and additional information on the SQL control statements see the following topics:

- "assignment-statement" on page 918
- "CALL statement" on page 920
- "CASE statement" on page 922
- "compound-statement" on page 924
- "FOR statement" on page 933

References to SQL parameters and SQL variables

SQL parameters and SQL variables can be referenced anywhere in the statement where an expression or variable can be specified. Host variables cannot be specified in SQL routines. 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. 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 924, 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.

  In DB2 for LUW, column names specified in CREATE TABLE, CREATE VIEW, and DECLARE GLOBAL TEMPORARY TABLE statements found in the routine body before the reference to the name are included in the search for the name.
- If not found as a column, the name will 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,[137] 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, global variable, 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. If all the tables and views exist at the current server, it will be assumed to be a global variable. Otherwise, an error is returned.

In DB2 for LUW, if the table or view does not exist at the server and is not created or declared before the reference, an error is returned.

The name of an SQL parameter or SQL variable 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:

137. Which can happen if they are declared in different compound statements.

- In the SET PATH or SET SCHEMA statements, the name is checked as an SQL variable name or an SQL parameter name. If an SQL variable or SQL parameter by that name is not found, the name is assumed to be an identifier.
- In the CONNECT, DISCONNECT, RELEASE, and SET CONNECTION statements, 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 924, 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 924, 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 routine 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 56. 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.

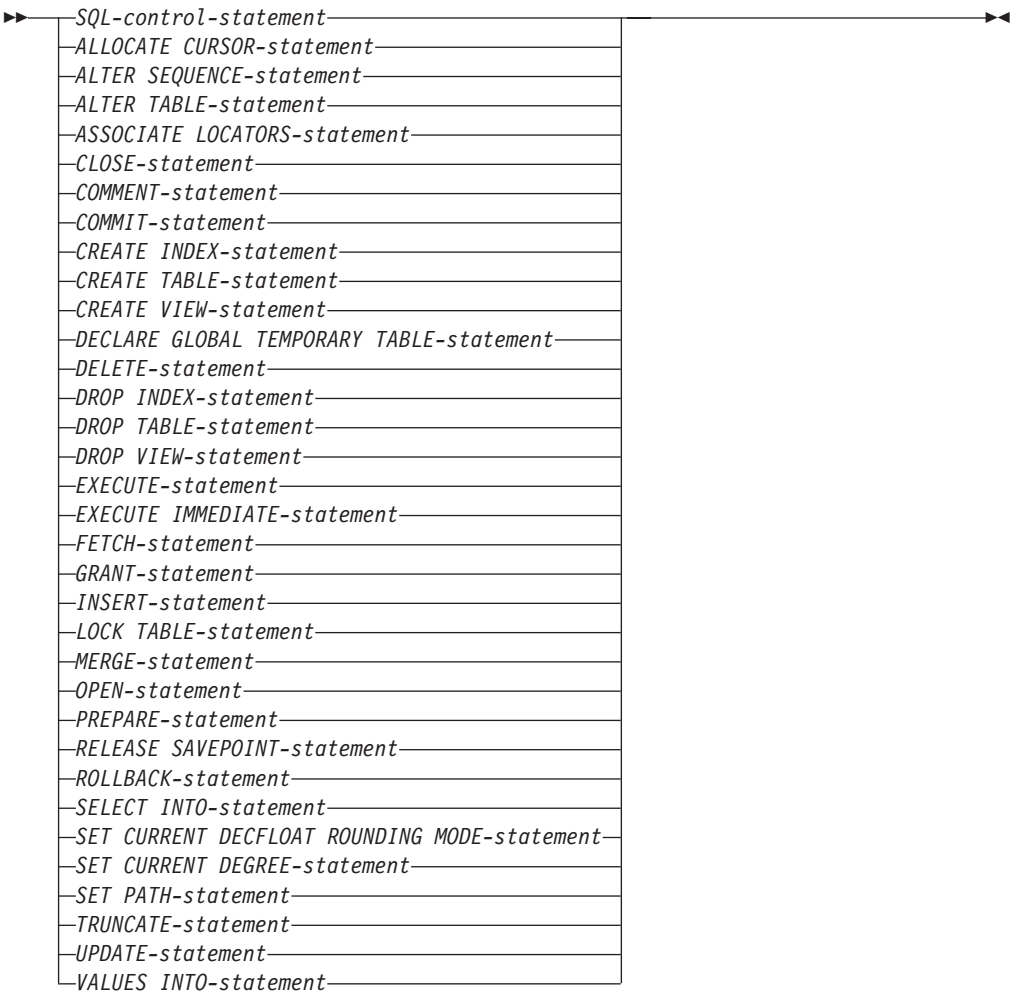
Table 56. Summary of 'Name' Scoping in Nested Compound Statements (continued)

Type of name	Must be unique within...	Qualification allowed?	Can be referenced within...
label	the compound statement that defined the label, including any definitions in compound statements that are nested within that compound statement and the name of the SQL routine that contains the 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



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 924 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:

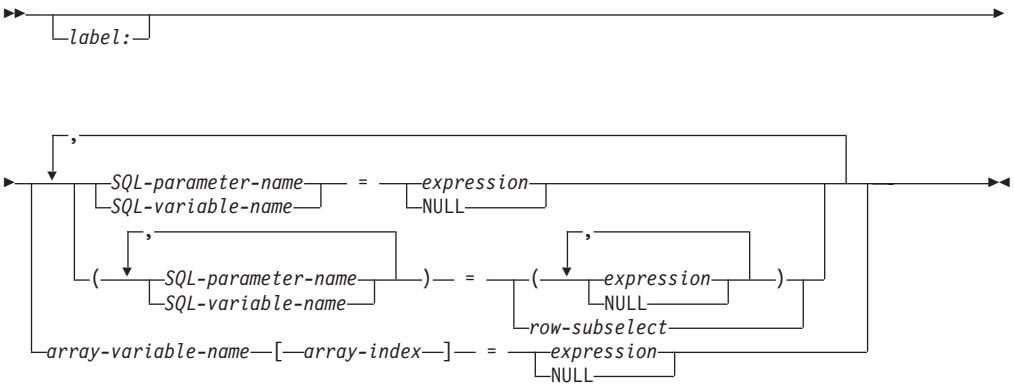
- Issue a GET DIAGNOSTICS statement to request the condition information. See "GET DIAGNOSTICS statement" on page 935.
- Test the SQL variables SQLSTATE and SQLCODE.

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 terminates with the error condition.

assignment-statement

The assignment statement assigns a value to an SQL parameter or an SQL variable.

Syntax



Description

label
Specifies the label for the *assignment-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 913.

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 statement. For an explanation of references to SQL parameters, see "References to SQL parameters and SQL variables" on page 909.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables can only be declared in a *compound-statement* and must be declared before they are used. For an explanation of references to SQL variables, see "References to SQL parameters and SQL variables" on page 909.

expression **or NULL**
Specifies the expression or value that is the source for the assignment.

row-subselect
A subselect that returns a single result row. The result column values are assigned to the corresponding SQL variable or parameter. If the result of the subselect 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.

[*array-index***]**
Numeric expression that specifies which element in the array will be the target of the assignment. The array index 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.

Notes

Assignment rules: Assignments in the assignment statement must conform to the SQL assignment rules as described in "Assignments and comparisons" on page 77. If assigning to a string variable, storage assignment rules apply.

Assignments involving SQL parameters: An IN parameter can appear on the left or right side in 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 in an *assignment-statement*. If used without first being assigned a value, the value is undefined. 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.

| **Multiple SQL parameter or SQL variable assignments:** If more than one SQL parameter or SQL variable is specified as the target of the *assignment-statement*, the targets of the *assignment-statement* are completely evaluated before the assignments are performed. Thus, references to an SQL parameter or SQL variable in a target expression is always the value of the SQL parameter or SQL variable prior to any assignment.

Considerations for SQLSTATE and SQLCODE SQL variables: Assignment to these variables is not prohibited. However, it is not recommended as assignment does not affect the diagnostic area or result in the activation of condition handlers. Whether processing an assignment to these SQL variables causes the specified values for the assignment to be overlayed with the SQL return codes returned from executing the statement that does the assignment is product specific.

Examples

Example 1: Increase the SQL variable `p_salary` by 10 percent.

```
SET p_salary = p_salary * 1.10
```

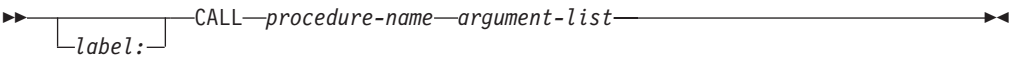
Example 2: Set SQL variable `p_salary` to the null value.

```
SET p_salary = NULL
```

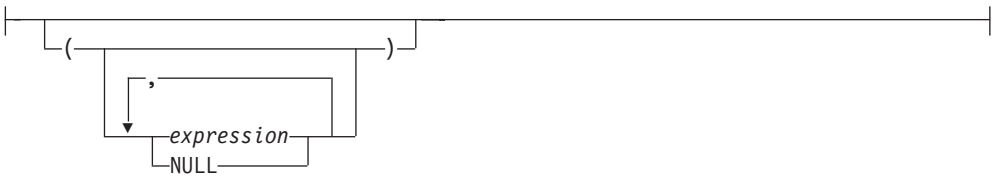
CALL statement

The CALL statement invokes a procedure. For additional details, see "CALL" on page 584.

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 913.

procedure-name
Identifies the procedure to call. The *procedure-name* must identify a procedure that exists at the current server and the procedure must be defined as an SQL procedure or a LANGUAGE C external procedure.

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 CREATE PROCEDURE) as OUT or INOUT must be specified as either a *SQL-variable-name* or a *SQL-parameter-name*.

The number of arguments specified must be the same as the number of parameters of a procedure defined at the current server with the specified *procedure-name*.

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.

expression
An *expression* of the type described in "Expressions" on page 130, that does not include an aggregate function or column name.

NULL
Specifies a null value as an argument to the procedure.

Notes

Related information: See "CALL" on page 584 for more information.

Examples

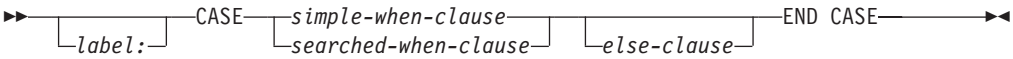
Call procedure *proc1* and pass SQL variables as parameters.

```
CALL proc1(v_empno, v_salary)
```

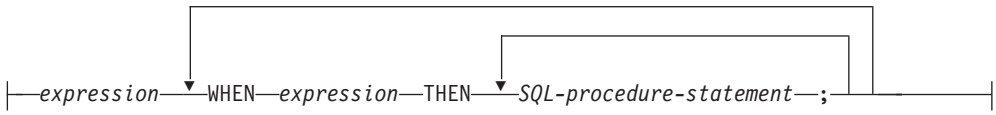
CASE statement

The CASE statement selects an execution path based on multiple conditions.

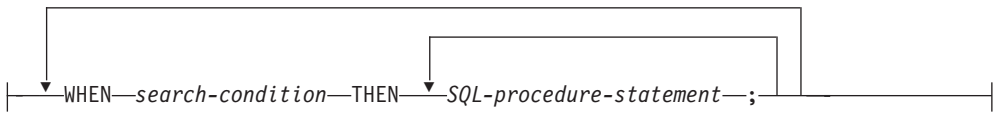
Syntax



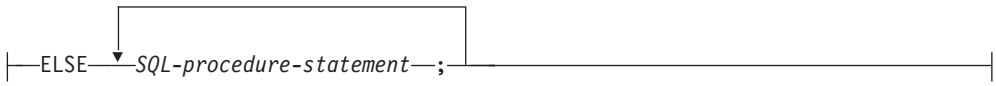
simple-when-clause:



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 913.

simple-when-clause
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. 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 returned at run time, 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 916.

Notes

Nesting 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.

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
```

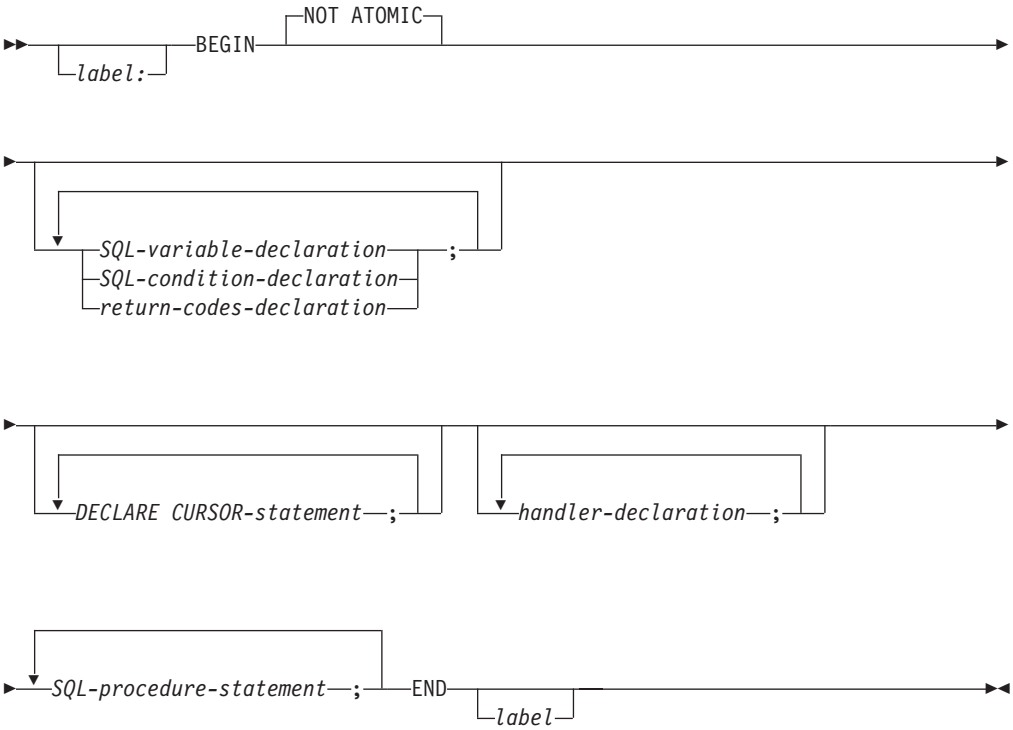
Example 2: The following example shows how to do this using the syntax for a *searched-when-clause*:

```
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
```

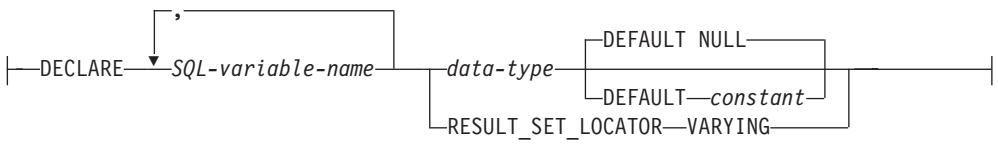
compound-statement

A compound statement groups other statements together in an SQL procedure. A compound statement allows the declaration of SQL variables, cursors, and condition handlers.

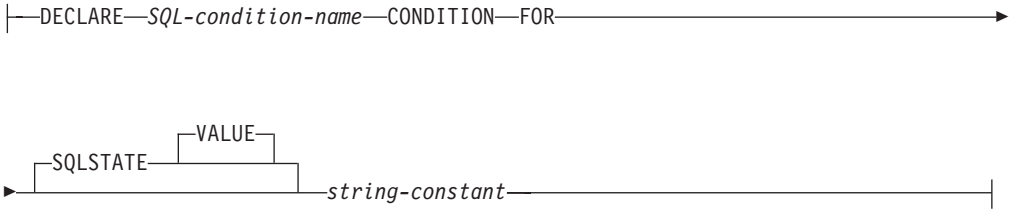
Syntax



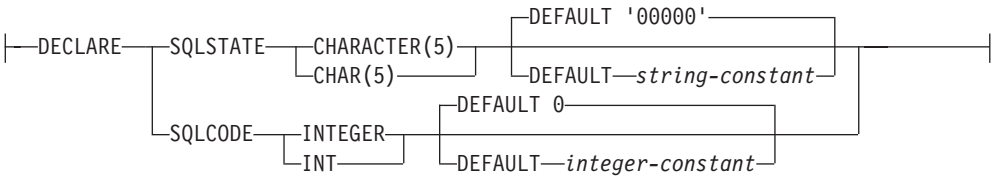
SQL-variable-declaration:



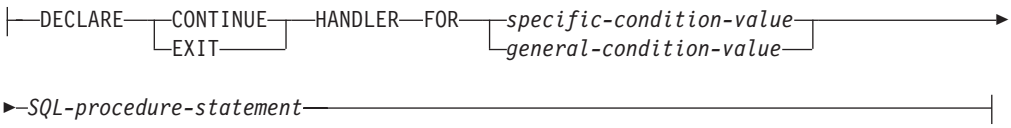
SQL-condition-declaration:



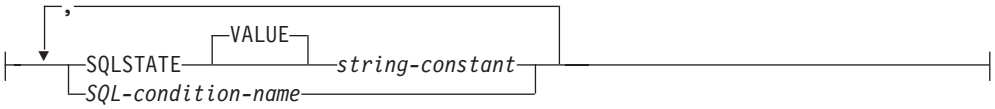
return-codes-declaration:



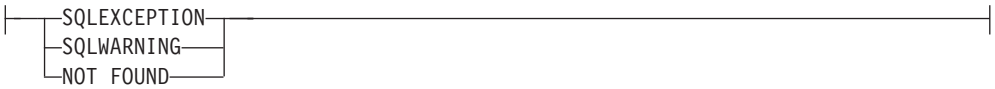
handler-declaration:



specific-condition-value:

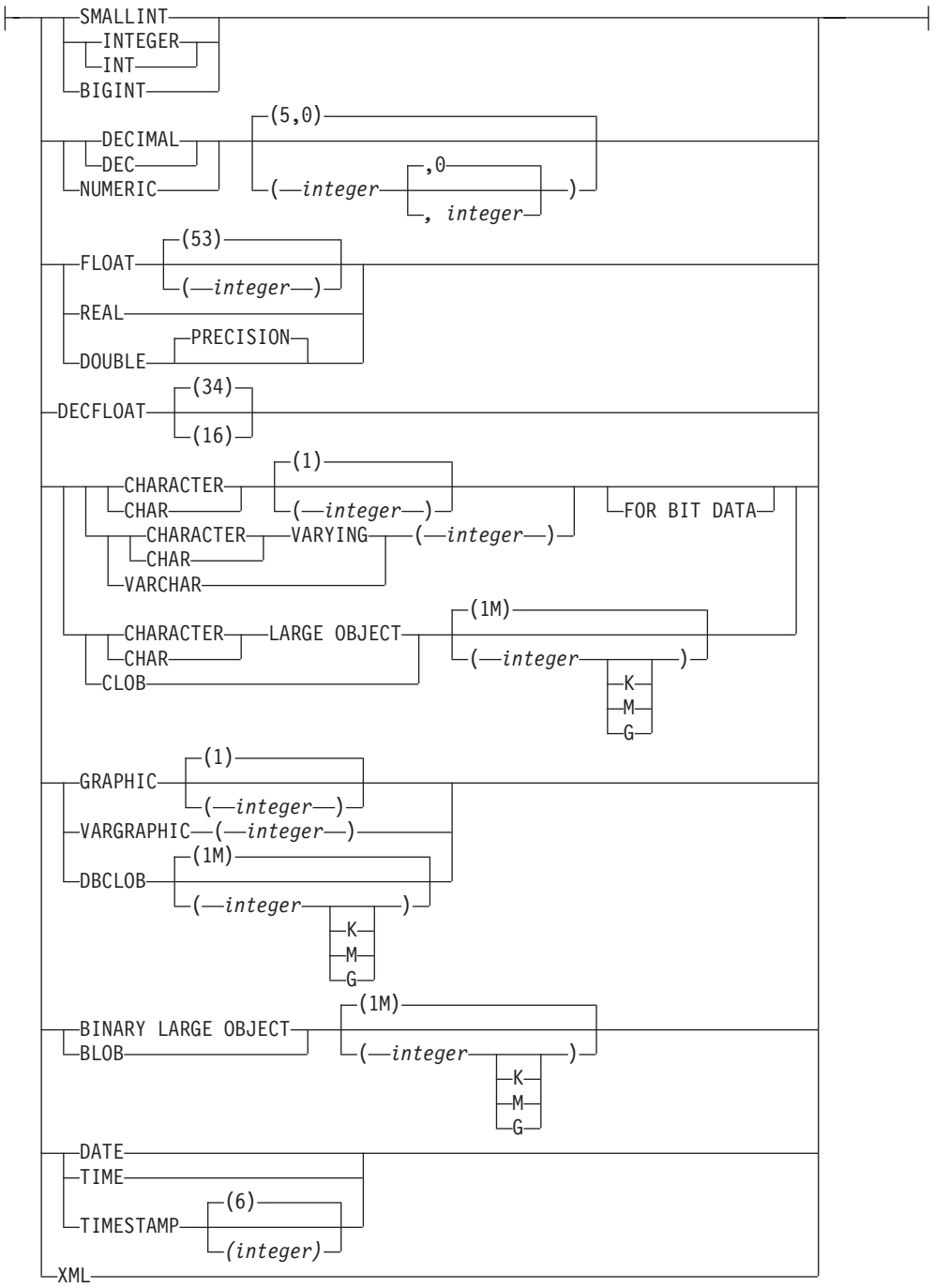


general-condition-value:



data-type:

built-in-type:



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 913.

NOT ATOMIC

NOT ATOMIC indicates that an unhandled exception condition within the *compound-statement* does not cause the *compound-statement* to be rolled back.

SQL-variable-declaration
Declares an SQL 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 909 for how SQL variable names are resolved when there are columns with the same name involved in a statement. Do not begin SQL variable names with 'SQL'. For an explanation of references to SQL variables, see "References to SQL parameters and SQL variables" on page 909.

data-type
Specifies the data type of the SQL variable. Refer to "Data types" on page 56 for a description of SQL data types.

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 77. The SQL variable is initialized when the SQL procedure is called. 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. Only DEFAULT NULL can be explicitly specified if *array-type-name* is 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 that is associated with the condition. The *string-constant* must be specified as five characters and the SQLSTATE class (the first two characters) must not be '00'.

return-codes-declaration
Declares special variables named SQLSTATE and SQLCODE. These variables are automatically set to the SQLSTATE and SQLCODE values for the first condition in the diagnostics area after executing an SQL statement other than GET DIAGNOSTICS or an empty compound statement. Both the SQLSTATE and SQLCODE variables can only be declared in the outermost *compound-statement* of the SQL procedure.

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 as assignment does not affect the diagnostic area or result in the activation of condition handlers. It is product specific whether processing an assignment to these SQL variables causes the specified values for the assignment to be overlayed with the SQL return codes returned from executing the statement that does the assignment. The SQLCODE and SQLSTATE variables cannot be set to NULL.

DECLARE CURSOR-statement
Declares a cursor in the procedure body. Each cursor must have a unique name within the *compound-statement* in which it is declared. The cursor can only be referenced from within the *compound-statement* in which it is declared, including any compound statements that are nested within that *compound-statement*.

Use an OPEN statement to open the cursor, a FETCH statement to read a row using the cursor, and a CLOSE statement to close the cursor. If the cursor 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 for the cursor 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 on declaring a cursor, refer to "DECLARE CURSOR" on page 749.

handler-declaration
Specifies a *handler*, an *SQL-procedure-statement* to execute when an exception or completion condition occurs in the *compound-statement*. *SQL-procedure-statement* is a statement that executes when the handler receives control. A *handler-declaration* can only include a single *SQL-procedure-statement*.

A condition handler declaration cannot reference the same condition value or SQLSTATE value more than one time. It cannot reference an SQLSTATE value and a condition name that represent the same SQLSTATE value.

When two or more condition handlers are declared in a compound statement, no two condition handler declarations can specify the same:

- general condition category
- 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-statement*s that follow the *handler-declaration*s within the *compound-statement* in which it is declared, including any nested compound statements.

There are two types of condition handlers:

CONTINUE
Specifies that after the condition handler is activated and completes successfully, control is returned to the SQL statement that follows the statement that raised the condition. However, if the condition is an error

condition and it was encountered while evaluating a search condition, as in a CASE, FOR, IF, REPEAT or WHILE statement, control returns to the statement that follows the corresponding END CASE, END FOR, END IF, END REPEAT, or END WHILE.

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 handler.

The condition that causes the handler to be invoked are defined in the *handler-declaration* as follows.

SQLSTATE VALUE *string*
Specifies that the handler is invoked when the specific SQLSTATE occurs. The first two characters of the SQLSTATE value 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'.

If the *SQL-procedure-statement* specified in the handler is either a SIGNAL or RESIGNAL statement with an exception SQLSTATE, the compound-statement will exit with the specified exception even if this handler or another handler in the same *compound-statement* specifies CONTINUE, since these handlers are not in the scope of this exception. If the *compound-statement* is nested in another *compound-statement*, handlers in the higher level compound-statement may handle the exception because those handlers are within the scope of the exception.

Notes

Nesting compound statements: Compound statements can be nested. Nested compound statements can be used to scope handlers and cursors to a subset of the statements in a procedure. This can simplify the processing done for each SQL procedure statement. Nested compound statements enables the use of a compound statement within the declaration of a condition handler.

Condition handlers: Condition handlers in SQL procedures are similar to WHENEVER statements that are 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 the result of an exception, a warning, or a not found condition that is returned by DB2 for the processing of an SQL statement. The condition that activates the handler can also be the result of a SIGNAL or RESIGNAL statement that is 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. For example, the scope of a condition handler declaration H is the list of *SQL-procedure-statements* that follow the condition handler declarations that are contained within the compound statement in which H appears. This means that the scope of H does not include the statements that are 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 that are declared in the same compound statement, H1 will not handle conditions that arise in the body of H2, and H2 will not handle conditions that arise in the body of H1.

The declaration of a condition handler specifies the condition that activates it, the type of condition handler (CONTINUE or EXIT), and the handler action. The type of condition handler determines to where control is returned after the handler action successfully completes.

Condition handler activation: 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.

In a routine with nested compound statements, condition handlers that could handle a specific condition might 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 the nesting level could handle the condition, the condition handler that is activated is the most appropriate handler that is declared in that compound statement.

The most appropriate handler is the condition handler that most closely matches the SQLSTATE or the exception or completion condition. For a given compound statement, when both a specific handler for a condition and a general handler are declared that address the same condition, the specific handler takes precedence over the general handler.

For example, if the innermost compound statement declares a specific handler for SQLSTATE '22001', as well as a general handler for SQLEXCEPTION, the specific handler for SQLSTATE '22001' is the most appropriate handler when 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 or EXIT handler) determines to 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 that is encountered in the handler action, that condition handler is activated. Otherwise, the condition that is 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 that contains the failing statement is terminated with an unhandled exception condition.
- If the unhandled condition is a warning or is a 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* that is 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 when 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 a compound statement is nested in another compound statement, condition handlers in the higher level compound statement can 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 a SQL statement (such as CONNECT or DESCRIBE) that does not allow an indicator variable, an error is returned.

Effect on open cursors: At the end of the compound statement, all open cursors that were declared in that compound statement, except cursors that are used to return result sets, are closed.

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.

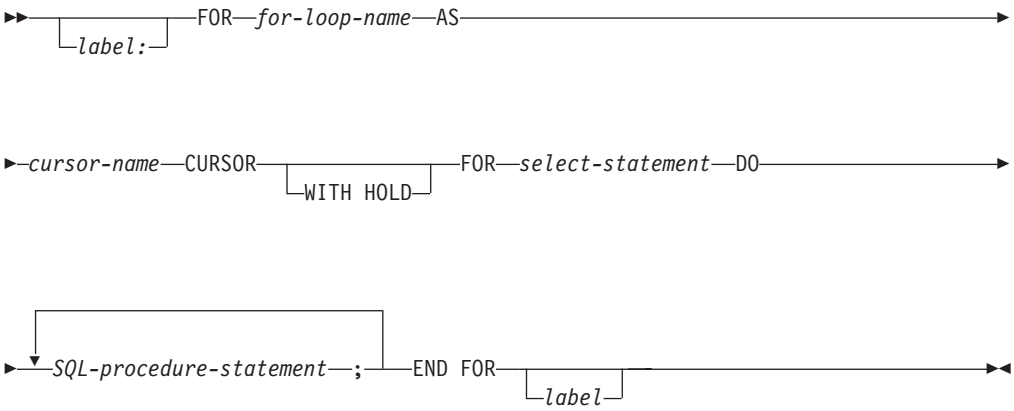
```
CREATE PROCEDURE DEPT_MEDIAN
   (IN  deptNumber   SMALLINT,
    OUT medianSalary DOUBLE)
   LANGUAGE SQL
   BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
```

```
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 913.

for-loop-name
Specifies a label for the implicit *compound-statement* generated to implement the FOR statement. It follows the rules for the label of a *compound-statement* except that it cannot be used with any 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*. The *for-loop-name* cannot be the same as another label within the same scope. For more information, see "References to SQL labels" on page 913.

cursor-name
Names the cursor that is used to select rows from the result table from the SELECT statement.

WITH HOLD
Prevents the cursor from being closed as a consequence of a commit operation. For more information, "DECLARE CURSOR" on page 749.

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 the 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 routine 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.

Examples

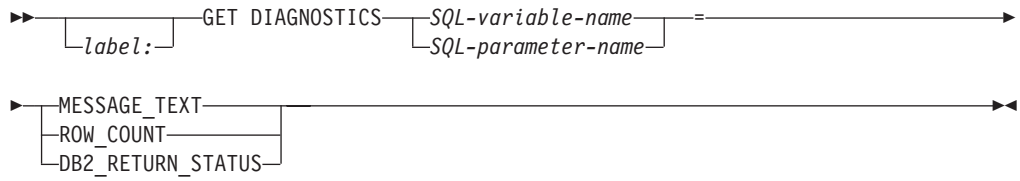
In the following example, the for-statement is used to iterate over the entire `employee` table. For each row in the table, the SQL variable `fullname` is set to the last name of the employee, followed by a comma, the first name, a blank space, and the middle initial. Each value for `fullname` is inserted into table `tnames`.

```
BEGIN
   DECLARE fullname CHAR(40);
   FOR v1 AS
      SELECT firstnme, midinit, lastname FROM employee
         DO
         SET fullname = lastname || ',' || firstnme ||' ' || midinit;
         INSERT INTO tnames VALUE (fullname);
   END FOR
END
```

GET DIAGNOSTICS statement

The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed.

Syntax



Description

label
Specifies the label for the GET DIAGNOSTICS 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 913.

SQL-variable-name
Identifies the SQL variable that is the assignment target. The data type of the SQL variable must be compatible with the data type as specified in Table 57 for the specified diagnostic item. For an explanation of references to SQL variables, see "References to SQL parameters and SQL variables" on page 909.

SQL-parameter-name
Identifies the SQL parameter that is the assignment target. The data type of the SQL parameter must be compatible with the data type as specified in Table 57 for the specified diagnostic item. For an explanation of references to SQL parameters, see "References to SQL parameters and SQL variables" on page 909.

MESSAGE_TEXT
Identifies the message text of the error, warning, or successful completion returned from the previous SQL statement that was executed.

ROW_COUNT
Specifies that the number of rows associated with the previous SQL statement that was executed is to be returned in the identified SQL variable or SQL parameter. If the previous SQL statement is a DELETE, INSERT, MERGE, 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.

DB2_RETURN_STATUS
Specifies that the status value returned from the previous CALL statement is to be returned in the identified SQL variable or SQL parameter. 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 951.

Table 57. Data Types for GET DIAGNOSTICS Items

Item Name	Data Type
MESSAGE_TEXT	VARCHAR(32740)
ROW_COUNT	DECIMAL(31,0)

Table 57. Data Types for GET DIAGNOSTICS Items (continued)

Item Name	Data Type
DB2_RETURN_STATUS	INTEGER

Notes

Effect of statement: The GET DIAGNOSTICS statement does not change the contents of the diagnostics area.

Examples

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 an SQL 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 = DB2_RETURN_STATUS;
        IF RETVAL <> 0 THEN
           ...
           LEAVE A1;
        ELSE
           ...
        END IF;
      END A1
```

GOTO statement

The GOTO statement is used to branch to a user-defined label within an SQL routine.

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 913.

label2
Specifies a labeled statement where processing is to continue. Neither the labeled statement nor the GOTO statement can appear in a *handler-declaration*.

- 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 in a *compound-statement*, *label2* must be defined inside the same *compound-statement*, excluding a nested FOR statement or nested *compound-statement*.
- If the GOTO statement is defined in a handler, *label2* must be defined in the same handler.
- If the GOTO statement is defined outside of a handler, *label2* must not be defined within a 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 those statements are declared to return result sets. If the GOTO statement transfers control out of any directly or indirectly nested compound statements, all open cursors that are declared in those nested compound statements are also closed, unless those cursors are declared to return result sets.

Examples

In the following compound statement used in a CREATE PROCEDURE statement, the parameters *rating* and *v_empno* are passed into the procedure, which then returns the output parameter *return_parm* as a date duration. If the employee's time

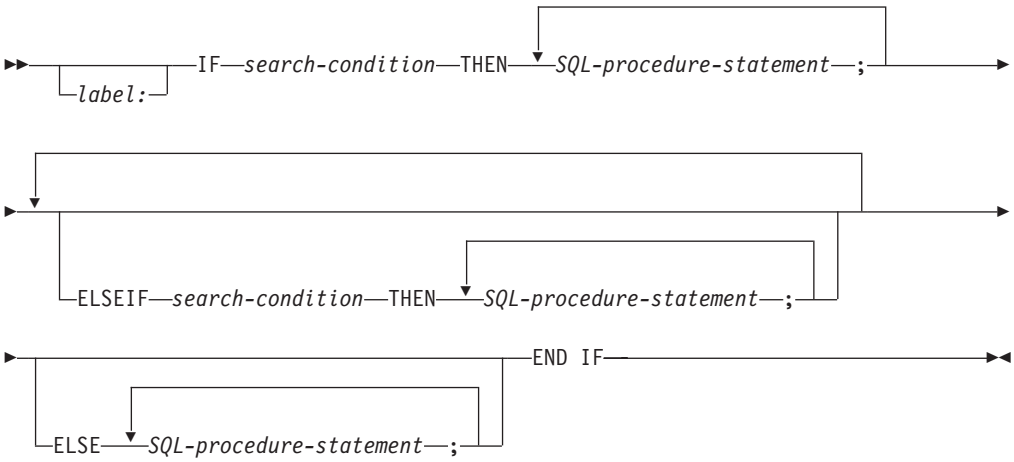
in service with the company is less than 6 months, the GOTO statement transfers control to the end of the procedure, and *new_salary* is left unchanged.

```
CREATE PROCEDURE adjust_salary
   (IN v_empno CHAR(6),
    IN rating INTEGER,
    OUT return_parm DECIMAL (8,0))
   LANGUAGE SQL
   MODIFIES SQL DATA
   BEGIN
      DECLARE new_salary DECIMAL(9,2);
      DECLARE service DECIMAL(8,0);
        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 913.

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 be executed if the preceding *search-condition* is true.

Examples

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.

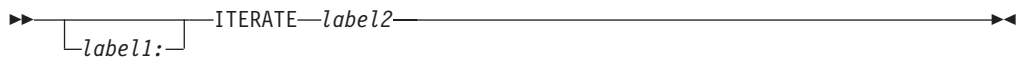
```
CREATE PROCEDURE UPDATE_SALARY_IF
   (IN employee_number CHAR(6),
    INOUT rating SMALLINT)
   LANGUAGE SQL
   BEGIN
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE EXIT HANDLER FOR not_found
         SET rating = -1;
      IF rating = 1
         THEN UPDATE employee
         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 labeled loop.

Syntax



Description

label1
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 913.

label2
Specifies the label of the FOR, LOOP, REPEAT, or WHILE statement to which the flow of control is passed. *label2* 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, *label2* must be defined in that condition handler.
- If the ITERATE statement is not in a condition handler, *label2* must not be defined in a condition handler.
- If the ITERATE statement is in a FOR statement, *label2* must be that label on that FOR statement, or the label must be defined inside that FOR statement.

Examples

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 transfers program control out of a FOR, LOOP, REPEAT, WHILE or compound statement.

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 913.

label2
Specifies the label of the compound, FOR, LOOP, REPEAT, or WHILE statement to exit.

Notes

Effect on open cursors: When a LEAVE statement transfers control out of a compound statement, all open cursors in the compound statement, except cursors that are used to return result sets, are closed.

Examples

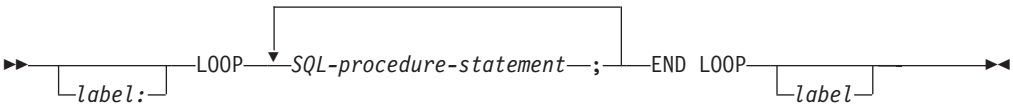
This example contains a loop that fetches data for cursor *c1*. If the value of SQL variable *at_end* is not zero, the LEAVE statement transfers control out of the loop.

```
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 beginning label is specified, that label can be specified on the LEAVE statement. If the ending label is specified, a matching beginning label must be specified. 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 913.

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 the SQLSTATE and SQLCODE SQL 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 successful completion of that statement. When the LOOP statement iterates, the SQLSTATE and SQLCODE values reflect the result of the last SQL statement that is 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.

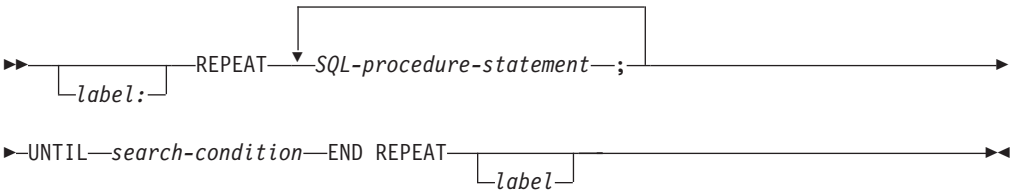
```
CREATE PROCEDURE LOOP_UNTIL_SPACE(OUT counter INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER DEFAULT 0;
      DECLARE v_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
            FROM employee;
      DECLARE EXIT 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
```

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 beginning label is specified, that label can be specified on the LEAVE statements. If an ending label is specified, a matching beginning label also must be specified. 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 913.

SQL-procedure-statement
Specifies an SQL statement to be executed within the REPEAT loop.

search-condition
The *search-condition* is evaluated after each execution of the REPEAT loop. If the condition is true, the 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 the 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.

Examples

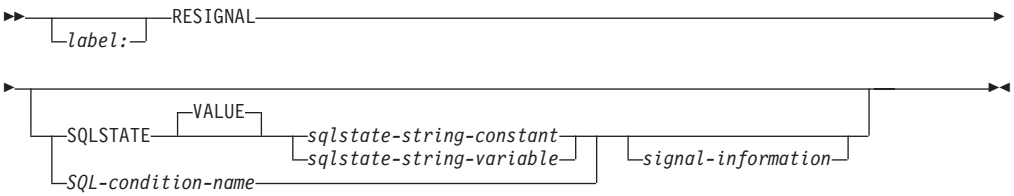
A REPEAT statement fetches rows from a table until the *not_found* condition handler is invoked.

```
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
```

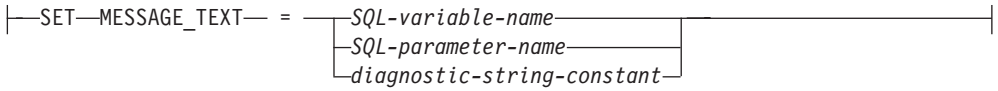
RESIGNAL statement

The RESIGNAL statement is used within a handler to resignal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

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 913.

SQLSTATE VALUE
Specifies the SQLSTATE that will be returned. 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 upper case letters ('A' through 'Z') without diacritical marks
- 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 specified variable must be of type CHAR(5).

SQL-condition-name
Specifies the name of a condition that will be returned. The *SQL-condition-name* must be declared within the *compound-statement*.

SET MESSAGE_TEXT
Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA. If the actual string is longer than 70 bytes, it is truncated without warning.

SQL-variable-name
Identifies an SQL variable, declared within the *compound-statement*, that contains the message text. The SQL variable must be defined as a CHAR or

VARCHAR data type. For an explanation of references to SQL variables, see "References to SQL parameters and SQL variables" on page 909.

SQL-parameter-name
Identifies an SQL parameter, defined for the procedure, that contains the message text. The SQL parameter must be defined as a CHAR or VARCHAR data type. For an explanation of references to SQL parameters, see "References to SQL parameters and SQL variables" on page 909.

diagnostic-string-constant
Specifies a character string constant that contains the message text.

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 on SQLSTATEs, see Chapter 13, "SQLSTATE values—common return codes," on page 997.

Processing a RESIGNAL statement:

- If the RESIGNAL statement is specified without an SQLSTATE clause or a *SQL-condition-name*, the identical condition that activated the handler is returned.
- If a RESIGNAL statement is issued, and an SQLSTATE or *SQL-condition-name* was specified, the SQLCODE returned is based on the SQLSTATE value as follows:
  - If the specified SQLSTATE class is either '01' or '02', a warning or not found is returned and the SQLCODE is set to +438
  - Otherwise, an exception is returned and the SQLCODE is set to -438.

Handlers and RESIGNAL statement:

- When the SQLSTATE or condition indicates an exception condition (SQLSTATE class other than '01' or '02'):
  - If an active handler exists for the specified SQLSTATE, condition or for SQLEXCEPTION, 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.
- When the SQLSTATE or condition indicates a warning condition (SQLSTATE class '01'):
  - If an active handler exists for the specified SQLSTATE, condition or for SQLWARNING, then the warning 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'), or the specified SQLSTATE or condition; the warning is not handled and processing continues with the next statement.

  - Otherwise, the warning is not handled and processing continues with the next statement.
- When the SQLSTATE or condition indicates a not found condition (SQLSTATE class '02'):
  - If an active handler exists for the specified SQLSTATE, condition or for NOT FOUND, then the 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 NOT FOUND (if the SQLSTATE class is '02'); the warning or not found condition is not handled and processing continues with the next statement.
  - Otherwise, the not found condition is not handled and processing continues with the next statement.

Examples

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
   CONTAINS SQL
   BEGIN
      DECLARE overflow CONDITION FOR SQLSTATE '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 is used to return from the routine. For an SQL function, it returns the result of the function. 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 913.

expression
Specifies the 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 77. An aggregate function, a user-defined function that is sourced on an aggregate function, or an OLAP specification must not be specified for a RETURN statement in an SQL scalar function. For a scalar function, *expression* cannot contain a scalar fullselect. See "Expressions" on page 130 for information on expressions.
- If the routine is a procedure, the data type of *expression* must be INTEGER. If *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 procedure, NULL must not be specified.

Notes

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 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 or equal to zero, the specified target for DB2_RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of zero.
  - 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 CLI 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.

Examples

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;
   FAIL: RETURN -200;
END
```

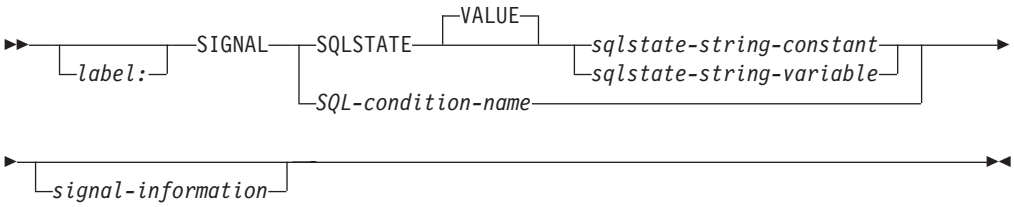
Example 2: Define a scalar function that returns the tangent of a value using the existing sine and cosine functions.

```
CREATE FUNCTION mytan (x DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(x)/COS(x)
```

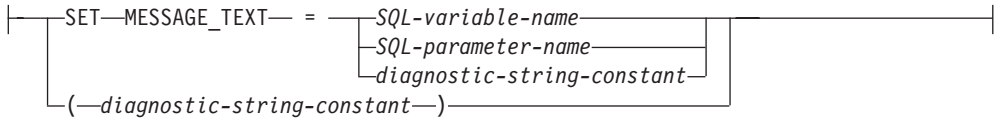
SIGNAL statement

The SIGNAL statement is used to signal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

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 913.

SQLSTATE VALUE
Specifies the SQLSTATE that will be returned. 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 upper case letters ('A' through 'Z') without diacritical marks
- 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 specified variable must be of type CHAR(5). It cannot be a global variable.

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 MESSAGE_TEXT
Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA. If the actual string is longer than 70 bytes, it is truncated without warning.

SQL-variable-name
Identifies an SQL variable, declared within the *compound-statement*, that contains the message text. The SQL variable must be defined as a CHAR or VARCHAR data type. For an explanation of references to SQL variables, see "References to SQL parameters and SQL variables" on page 909. It cannot be a global variable.

SQL-parameter-name
Identifies an SQL parameter, defined for the procedure, that contains the message text. The SQL parameter must be defined as a CHAR or VARCHAR data type. For an explanation of references to SQL parameters, see "References to SQL parameters and SQL variables" on page 909.

diagnostic-string-constant
Specifies a character string constant that contains the message text. If the string is longer than 70 bytes, it will be truncated without warning.

(*diagnostic-string-constant***)**
Specifies a character string constant that contains the message text. If the string is longer than 70 bytes, it will be truncated without warning. Within the triggered action of a CREATE TRIGGER statement, the message text can only be specified using this syntax:

```
SIGNAL SQLSTATE sqlstate-string-constant (diagnostic-string-constant);
```

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 on SQLSTATEs, see Chapter 13, "SQLSTATE values—common return codes," on page 997.

Processing a SIGNAL statement: If a SIGNAL statement is issued, the SQLCODE returned is based on the SQLSTATE value as follows:

- If the specified SQLSTATE class is either '01' or '02', a warning or not found is returned and the SQLCODE is set to +438
- Otherwise, an exception is returned and the SQLCODE is set to -438.

Handlers and SIGNAL statement:

- When the SQLSTATE or condition indicates an exception condition (SQLSTATE class other than '01' or '02'):
  - If an active handler exists for the specified SQLSTATE, condition or for SQLEXCEPTION, 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.
- When the SQLSTATE or condition indicates a warning condition (SQLSTATE class '01'):

  - If an active handler exists for the specified SQLSTATE, condition or for SQLWARNING, then the warning 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'), or the specified SQLSTATE or condition; the warning is not handled and processing continues with the next statement.
  - Otherwise, the warning is not handled and processing continues with the next statement.
- When the SQLSTATE or condition indicates a not found condition (SQLSTATE class '02'):
  - If an active handler exists for the specified SQLSTATE, condition or for NOT FOUND, then the 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 NOT FOUND (if the SQLSTATE class is '02'); the warning or not found condition is not handled and processing continues with the next statement.
  - Otherwise, the not found condition is not handled and processing continues with the next statement.

Examples

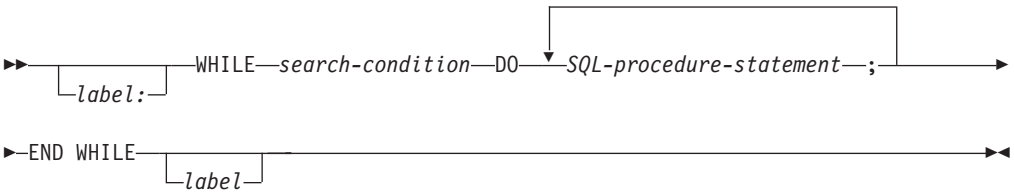
An SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
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 or group of statements while a specified condition is true.

Syntax



Description

label
Specifies the label for the WHILE statement. If the beginning label is specified, it can be specified in the LEAVE 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 913.

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 the 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 that search condition of that WHILE statement.

Examples

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
```

Chapter 9. SQL limits

The following tables describe certain SQL and database limits imposed by the IBM relational database products. Adhering to the most restrictive case can help the programmer design application programs that are easily portable.

Note:

- System storage limits may preclude the limits specified here. For example, see Byte Counts.
- 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.

138. As an application requester, DB2 for i can send an authorization name of up to 255 bytes.

139. Except for DECLARE CURSOR WITH RETURN or the EXEC SQL utility.

140. Individual host languages may vary.

141. The schema name can be up to 128 bytes for the schema name of all objects except distinct types.

Table 58. Identifier length limits

Identifier Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Longest authorization name	8	10[138]	128	8
Longest constraint name	128	128	128	128
Longest correlation name	128	128	128	128
Longest cursor name	128[139]	128	128	128
Longest external program name (string form)	1305	279	254	254
Longest external program name (unqualified form)	8	10	128	8
Longest host identifier[140]	128	128	255	128
Longest package version-ID	64	64	64	64
G Longest partition name	n/a	10	128	10
Longest savepoint name	128	128	128	128
Longest schema name	128	128	128[141]	128
Longest server name	16	18	8	18
Longest SQL condition name	128	128	128	128
Longest SQL label	128	128	128	128
Longest statement name	128	128	128	128
Longest unqualified alias name	128	128	128	128
Longest unqualified column name	30	128	128	30
Longest unqualified function name	128	128	128	128
Longest unqualified global variable name	128	128	128	128
Longest unqualified index name	128	128	128	128
Longest unqualified mask name	128	128	128	128
Longest unqualified package name	8	10	128	8
Longest unqualified permission name	128	128	128	128
Longest unqualified procedure name	128	128	128	128
Longest unqualified sequence name	128	128	128	128
Longest unqualified specific name	128	128	128	128
Longest unqualified SQL parameter name	128	128	128	128
Longest unqualified SQL variable name	128	128	128	128
Longest unqualified table and view name	128	128	128	128
Longest unqualified trigger name	128	128	128	128
Longest unqualified type name	128	128	128	128

Table 58. Identifier length limits (continued)

Identifier Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Longest XML element name, attribute name, prefix name, or processing instruction name specified in XMLELEMENT, XMLFOREST, XMLATTRIBUTES, XMLNAMESPACES or XMLPI	128	128	??	??
Longest XML element name, attribute name, prefix name, or processing instruction name for a parsed XML document	1000	1000	1000	1000
Longest XML schema location uniform resource identifier (URI)	1000	1000	1000	1000

142. The values shown are approximate.

Table 59. Numeric limits

Numeric Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Smallest SMALLINT value	-32 768	-32 768	-32 768	-32 768
Largest SMALLINT value	+32 767	+32 767	+32 767	+32 767
Smallest INTEGER value	-2 147 483 648	-2 147 483 648	-2 147 483 648	-2 147 483 648
Largest INTEGER value	+2 147 483 647	+2 147 483 647	+2 147 483 647	+2 147 483 647
Smallest BIGINT value	-9 223 372 036 854 775 808 for all columns			
Largest BIGINT value	+9 223 372 036 854 775 807 for all columns			
Largest decimal precision	31	63	31	31
Maximum exponent (E_{max}) for REAL values	75	38	38	38
Smallest REAL value[142]	$-7.2x10^{75}$	$-3.4x10^{38}$	$-3.4x10^{38}$	$-3.4x10^{38}$
Largest REAL value[142]	$+7.2x10^{75}$	$+3.4x10^{38}$	$+3.4x10^{38}$	$+3.4x10^{38}$
Minimum exponent (E_{min}) for REAL values	-79	-38	-37	-37
Smallest positive REAL value[142]	$+5.4x10^{-79}$	$+1.18x10^{-38}$	$+1.17x10^{-37}$	$+1.17x10^{-37}$
Largest negative REAL value[142]	$-5.4x10^{-79}$	$-1.18x10^{-38}$	$-1.17x10^{-37}$	$-1.17x10^{-37}$
Maximum exponent (E_{max}) for DOUBLE values	75	308	308	75
Smallest DOUBLE value[142]	$-7.2x10^{75}$	$-1.79x10^{308}$	$-1.79x10^{308}$	$-7.2x10^{75}$
Largest DOUBLE value[142]	$+7.2x10^{75}$	$+1.79x10^{308}$	$+1.79x10^{308}$	$+7.2x10^{75}$
Minimum exponent (E_{min}) for DOUBLE values	-79	-308	-307	-79
Smallest positive DOUBLE value[142]	$+5.4x10^{-79}$	$+2.23x10^{-308}$	$+2.23x10^{-307}$	$+5.4x10^{-79}$
Largest negative DOUBLE value[142]	$-5.4x10^{-79}$	$-2.23x10^{-308}$	$-2.23x10^{-307}$	$-5.4x10^{-79}$
Maximum exponent (Emax) for DECFLOAT(16) values	384	384	384	384
Smallest DECFLOAT(16) value[146]	$-9.999999999999999x10^{384}$ for all columns			
Largest DECFLOAT(16) value[146]	$9.999999999999999x10^{384}$ for all columns			
Minimum exponent (Emin) for DECFLOAT(16) values	-383	-383	-383	-383
Smallest positive DECFLOAT(16) value[146]	$1x10^{-383}$	$1x10^{-383}$	$1x10^{-383}$	$1x10^{-383}$
Largest negative DECFLOAT(16) value[146]	$-1x10^{-383}$	$-1x10^{-383}$	$-1x10^{-383}$	$-1x10^{-383}$
Maximum exponent (Emax) for DECFLOAT(34) values	6144	6144	6144	6144
Smallest DECFLOAT(34) value[146]	$-9.999999999999999999999999999999999x10^{6144}$ for all columns			
Largest DECFLOAT(34) value[146]	$9.999999999999999999999999999999999x10^{6144}$ for all columns			
Minimum exponent (Emin) for DECFLOAT(34) values	-6143	-6143	-6143	-6143
Smallest positive DECFLOAT(34) value[146]	$1x10^{-6143}$	$1x10^{-6143}$	$1x10^{-6143}$	$1x10^{-6143}$
Largest negative DECFLOAT(34) value[146]	$-1x10^{-6143}$	$-1x10^{-6143}$	$-1x10^{-6143}$	$-1x10^{-6143}$

143. If the column is NOT NULL, the limit is one more.

144. Further restricted by individual utilities and preprocessors.

145. The limit is 500 for sequences.

146. 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 60. String limits

String Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Maximum length of CHAR (in bytes)	255	32 765[143]	254	254
Maximum length of VARCHAR (in bytes)	32 704	32 739[143]	32 672	32 672
Maximum length of CLOB (in bytes)	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Maximum length of GRAPHIC (in double-byte characters)	127	16 382[143]	127	127
Maximum length of VARGRAPHIC (in double-byte characters)	16 352	16 369[143]	16 336	16 336
Maximum length of DBCLOB (in double-byte characters)	1 073 741 823	1 073 741 823	1 073 741 823	1 073 741 823
Maximum length of BLOB (in bytes)	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Maximum length of serialized XML (in bytes)	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Maximum length of character constant	32 704	32 740	32 672	32 672
Maximum length of a graphic constant[144]	16 352	16 370	16 336	16 336
Maximum length of a concatenated character string	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Maximum length of a concatenated graphic string	1 073 741 823	1 073 741 823	1 073 741 823	1 073 741 823
Maximum length of a concatenated binary string	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Maximum number of hexadecimal constant digits	32 704	32 762	32 672	32 672
Maximum length of catalog comments (in bytes)	254	2000[145]	254	254

Table 61. XML limits

XML Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Maximum length of an XML schema document (in bytes)	31 457 280	2 147 483 647	31 457 280	31 457 280
Maximum length of a parsed XML entity	Storage	Storage	??	??

147. Shown in ISO format.

Table 62. Datetime limits for IBM SQL and all IBM relational database products

Datetime Limits[147]	DB2 SQL and All IBM Relational Database Products
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

148. If the table is a dependent table, the limit is 749.

149. Row size may be further restricted by the page size of the table space.

150. Less for external routines with PARAMETER STYLE DB2SQL or PARAMETER STYLE JAVA.

151. The numbers shown are architectural limits and approximations. The practical limits may be less.

152. The longest index key is actually the number provided in the table minus the number of columns that allow nulls.

153. The maximum can be less depending on index options.

154. This is an approximate guideline. In a complex SELECT, the number of tables that can be joined may be significantly less.

155. In REXX, the maximum number of prepared statements is 100. Of these, no more than 50 can be declared cursors with the WITH HOLD clause, and no more than 50 can be declared cursors without the WITH HOLD clause.

156. Further limited by the presence of nested procedures and functions.

Table 63. Database manager limits

Database Manager Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Relational Database				
Maximum number of schemas	storage	4740	storage	4740
Most tables in a relational database[151]	storage	storage	storage	storage
Schemas				
Maximum number of objects in a schema	storage	360 000[142]	storage	360 000[142]
Tables and Views				
Maximum number of columns in a table	750[148]	8000	1012	750
Maximum number of columns in a view	750	8000	5000	750
Maximum length of a row including all overhead	32 714[149]	32 766	32 677[149]	32 677
Maximum number of rows in a non-partitioned table[142]	4 278 190 080	4 294 967 288	128x10[10]	4 278 190 080
Maximum number of rows in a data partition[142]	4 278 190 080	4 294 967 288	128x10[10]	4 278 190 080
Maximum size of a non-partitioned table[151]	64 gigabytes	1.7 terabytes	512 gigabytes	64 gigabytes
Maximum size of a data partition[151]	64 gigabytes	1.7 terabytes	512 gigabytes	64 gigabytes
Maximum number of data partitions in a single partitioned table or partitioned index	4096	256	32767	256
Maximum number of table partitioning columns	64	120	16	16
Most tables referenced in a view or materialized query table	225[154]	256	storage	225
Constraints				
Maximum number of constraints on a table	storage	5000	storage	5000
Maximum number of columns in a UNIQUE constraint	64	120	64	64
Maximum combined length of columns in a UNIQUE constraint (in bytes)[152]	2000[153]	32 767	8192	2000
Maximum number of referencing columns in a foreign key	64	120	64	64
Maximum combined length of referencing columns in a foreign key (in bytes)[152]	2000[153]	32 767	8192	2000
Maximum length of a CHECK constraint (in bytes)	3800	statement	65 535	3800
Triggers				
Maximum number of triggers on a table	storage	300	storage	300

Table 63. Database manager limits (continued)

Database Manager Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Maximum number of nested trigger levels	16[156]	200	16	16
Indexes				
Maximum number of indexes on a table	storage	4000[142]	32 767 or storage	4000[142]
Maximum number of columns in an index key	64	120	64	64
Maximum length of an index key[152]	2000[153]	32 767	8192	2000
Maximum size of a non-partitioned index[151]	64 gigabytes	1.7 terabytes	16 384 gigabytes	64 gigabytes
Maximum size of a index partition[151]	64 gigabytes	1.7 terabytes	16 384 gigabytes	64 gigabytes
SQL				
Maximum length of an SQL statement (in bytes)	2 097 152	2 097 152	2 097 152	2 097 152
Maximum number of tables referenced in an SQL statement	225[154]	1000	storage	225[154]
Maximum number of variables and constants in an SQL statement	statement	4096	32767	4096
Maximum number of elements in a select list	750	8000[142]	1012	750
Maximum number of predicates in a WHERE or HAVING clause	statement	statement	statement	statement
Maximum number of columns in a GROUP BY clause	statement	total GROUP BY length	1012	120
Maximum total length of columns in a GROUP BY clause	16 000	3.5 gigabytes	32 677	16 000
Maximum number of columns in an ORDER BY clause	statement	total ORDER BY length	1012	1012
Maximum total length of columns in an ORDER BY clause	16 000	3.5 gigabytes	32 677	16 000
Maximum levels allowed for a subquery	224	256	storage	224
Routines				
Maximum number of parameters in a procedure	statement	1024[150]	32 767	1024
Maximum number of parameters in a function	statement	90	90	90
Applications				
Most host variable declarations in a precompiled program	storage	storage	storage	storage
Maximum length of a host variable value (in bytes)	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Maximum length of an MQ message CLOB value (in bytes)	1M	2M	1M	1M
Maximum length of an MQ message varying length value (in bytes)	4000	32000	32000	4000

Table 63. Database manager limits (continued)

Database Manager Limits	DB2 for z/OS	DB2 for i	DB2 for LUW	DB2 SQL
Most declared cursors in a program	storage	storage	storage	storage
Maximum number of cursors opened at one time	storage	storage	storage	storage
Maximum number of prepared statements	storage	storage	storage[155]	storage
Maximum length of an SQL path	2048	8843	2048	2048
Maximum length of a password	8	127	18	8
Maximum length of a password hint	32	32	32	32

Chapter 10. 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 972 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 974 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 976 provides information about the use of SQL statements when the application server is not the same as the application requester.

Actions allowed on SQL statements

Table 64 shows whether a specific SQL statement can be executed, issued interactively or prepared dynamically, or processed by the requester, the server, or the precompiler. The letter **Y** means *yes*.

Table 64. Actions allowed on SQL statements

SQL statement	Executable	Issued interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
ALLOCATE CURSOR[159]	Y		Y		
ALTER	Y	Y		Y	
ASSOCIATE LOCATORS[159]	Y		Y		
BEGIN DECLARE SECTION[158,159]					Y
CALL[157]	Y			Y	
CLOSE[158]	Y			Y	
COMMENT	Y	Y		Y	
COMMIT	Y	Y		Y	
CONNECT (type 1 and type 2)[158,159]	Y		Y		
CREATE	Y	Y		Y	
DECLARE CURSOR[158]					Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y		Y	
DELETE	Y	Y		Y	
DESCRIBE[158]	Y			Y	
DROP	Y	Y		Y	
END DECLARE SECTION[158,159]					Y
EXECUTE[158]	Y			Y	
EXECUTE IMMEDIATE[158]	Y			Y	
FETCH	Y			Y	
FREE LOCATOR[158,159]	Y			Y	
GRANT	Y	Y		Y	
INCLUDE[158,159]					Y
INSERT	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
MERGE	Y	Y		Y	
OPEN[158]	Y			Y	
PREPARE[158]	Y			Y	

157. The statement can be dynamically prepared, but only from a CLI, ODBC or JDBC driver that supports dynamic CALL statements.

158. This statement is not applicable in a Java program.

159. This statement is not supported in a REXX program.

Table 64. Actions allowed on SQL statements (continued)

SQL statement	Executable	Issued interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
REFRESH TABLE	Y	Y		Y	
RELEASE connection[158,159]	Y		Y		
RELEASE SAVEPOINT	Y	Y		Y	
RENAME	Y	Y		Y	
REVOKE	Y	Y		Y	
ROLLBACK	Y	Y		Y	
ROLLBACK TO SAVEPOINT	Y	Y		Y	
SAVEPOINT	Y	Y		Y	
SELECT INTO[159]	Y			Y	
SET CONNECTION[158,159]	Y		Y		
SET CURRENT DECFLOAT ROUNDING MODE	Y	Y		Y	
SET CURRENT DEGREE	Y	Y		Y	
SET ENCRYPTION PASSWORD	Y	Y		Y	
SET PATH	Y	Y		Y	
SET SCHEMA	Y	Y		Y	
SET transition-variable[160]	Y			Y	
SET variable	Y	Y [161]	Y		
SQL-control-statement	Y			Y	
TRUNCATE	Y	Y		Y	
UPDATE	Y	Y		Y	
VALUES[160]	Y			Y	
VALUES INTO[159]	Y			Y	
WHENEVER[158,159]					Y

160. This statement can only be used in the triggered action of a trigger.

161. The target of the SET variable statement must be a global variable.

SQL statement data access classification for routines

Table 65 indicates (using the letter Y) whether an SQL statement (specified in the first column) is allowed to execute in a routine with the specified SQL data access classification.

Table 65. SQL data access classification of SQL statements

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALLOCATE CURSOR			Y	Y
ALTER				Y
ASSOCIATE LOCATORS			Y	Y
BEGIN DECLARE SECTION	Y[162]	Y	Y	Y
CALL		Y[163]	Y[163]	Y[163]
CLOSE			Y	Y
COMMENT				Y
COMMIT		Y	Y	Y
CONNECT(type 1 and type 2)[164]				
CREATE				Y
DECLARE CURSOR	Y[162]	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE				Y
DELETE				Y
DESCRIBE			Y	Y
DROP				Y
END DECLARE SECTION	Y[162]	Y	Y	Y
EXECUTE		Y[165]	Y[165]	Y
EXECUTE IMMEDIATE		Y[165]	Y[165]	Y
FETCH			Y	Y
FREE LOCATOR		Y	Y	Y
GRANT				Y
INCLUDE	Y[162]	Y	Y	Y
INSERT				Y
LOCK TABLE		Y	Y	Y
MERGE				Y
OPEN			Y	Y
PREPARE		Y	Y	Y
REFRESH TABLE				Y
RELEASE connection[164]				
RELEASE SAVEPOINT				Y

162. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.

163. A CALL statement can only be used in a procedure defined as LANGUAGE SQL or LANGUAGE C.

164. Connection management statements are not allowed in any procedure execution contexts.

Table 65. SQL data access classification of SQL statements (continued)

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
RENAME				Y
REVOKE				Y
ROLLBACK		Y	Y	Y
ROLLBACK TO SAVEPOINT				Y
SAVEPOINT				Y
SELECT INTO			Y	Y
SET CONNECTION[164]				
SET CURRENT DECFLOAT ROUNDING MODE		Y	Y	Y
SET CURRENT DEGREE			Y	Y
SET ENCRYPTION PASSWORD		Y	Y	Y
SET PATH		Y	Y	Y
SET SCHEMA			Y	Y
SET transition-variable				
SQL-control-statement		Y	Y	Y
TRUNCATE				Y
UPDATE				Y
VALUES				
VALUES INTO			Y	Y
WHENEVER	Y[162]	Y	Y	Y

165. 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, MERGE, or DELETE.

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 the SQL described in this publication. Some of these extensions are product-specific, but many are already supported 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 via 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 requester, see Table 66
- for DB2 for i application requester, seeTable 67 on page 977
- for DB2 for LUW application requester, see Table 68 on page 977.

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 may mean that the function is available only if 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 66. DB2 for z/OS application requester

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 for 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		166	166
PREPARE with USING clause			R
ROLLBACK HOLD	R	R	R
SET CURRENT PACKAGESET			
SET host variable		R	R
SET TRANSACTION	R	R	R
Scrollable cursor statements	R	R	R

Table 66. DB2 for z/OS application requester (continued)

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 for LUW Application Server
UPDATE cursor - FOR UPDATE clause not specified			

Table 67. DB2 for i application requester

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 for LUW Application Server
COMMIT HOLD	R		R
DECLARE STATEMENT			
DECLARE TABLE			
DECLARE VARIABLE			
DESCRIBE TABLE			R
DESCRIBE with USING clause			R
DISCONNECT			
ROWID data types			R
DATALINK data types	R		R
BINARY and VARBINARY data types			R
Host declarations not documented in language specific appendices	[166]		[166]
PREPARE with USING clause			R
ROLLBACK HOLD	R		R
SET CURRENT PACKAGESET	R	R	R
SET host variable	R	R	R
SET TRANSACTION	R		R
Scrollable cursor statements	R		R
UPDATE cursor - FOR UPDATE clause not specified	R		

Table 68. DB2 for LUW application requester

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 for 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

166. The statement is supported if the application requester understands it.

Table 68. DB2 for LUW application requester (continued)

SQL Statement or Function	DB2 for z/OS Application Server	DB2 for i Application Server	DB2 for LUW Application Server
DISCONNECT			
ROWID data types	[167]	[167]	R
DATALINK data types	R	R	R
BINARY and VARBINARY data types	R	R	R
Host declarations not documented in language specific appendices	[166]	[166]	
PREPARE with USING clause	R	R	R
ROLLBACK HOLD	R	R	R
SET CURRENT PACKAGESET			
SET host variable	R	R	R
SET TRANSACTION	R	R	R
Scrollable cursor statements	R	R	R
UPDATE cursor - FOR UPDATE clause not specified	R		

167. The DB2 for LUW 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 "CONNECT (type 1)" on page 598
- CONNECT (type 2) is used for distributed unit of work. See "CONNECT (type 2)" on page 602

The following table summarizes the differences between CONNECT (type 1) and CONNECT (type 2) rules:

Table 69. CONNECT (type 1) and CONNECT (type 2) differences

Type 1 Rules	Type 2 Rules
CONNECT statements can only be executed when the application process 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.
G If the CONNECT statement fails because the G server name is not listed in the local G directory, the connection state of the G application process is product-specific. If a CONNECT statement fails because the application process is not in the connectable state, the connection status of the application process is unchanged. If a CONNECT statement fails for any other reason, the application process is placed in the unconnected state.	If a CONNECT statement fails, the current connection is unchanged and any subsequent SQL statements are executed by the current server.
CONNECT ends its only existing connection of the application process. Accordingly, CONNECT also closes any open cursors of the application process. (The only cursors that can possibly be open when CONNECT is successfully executed are those defined with the WITH HOLD option.)	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.[168]

Determining the CONNECT rules that apply

A program preparation option is used to specify the type of CONNECT that will G be performed by a program. The program preparation option is product-specific.

The CONNECT rules that apply to an application process are determined by the first CONNECT statement that is executed (successfully or unsuccessfully) by that application process:

- If it is a CONNECT (type 1), then CONNECT (type 1) rules apply and CONNECT (type 2) statements are invalid

168. In DB2 for z/OS, the SQLRULES(STD) bind option can be used to allow a CONNECT to the current server.

- If it is a CONNECT (type 2), then CONNECT (type 2) rules apply and CONNECT (type 1) statements are invalid.

Programs containing CONNECT statements that are precompiled with different CONNECT program preparation options cannot execute as part of the same application process. An error will occur when an attempt is made to execute the invalid CONNECT statement.

Connecting to application 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.

Chapter 11. SQLCA (SQL communication area)

An SQLCA is a set of variables that is updated at the end of the execution of every SQL statement. A program that contains executable SQL statements must provide exactly one SQLCA (unless a stand-alone SQLSTATE or a stand-alone SQLCODE variable is used instead), except in Java, where the SQLCA is not applicable.

The SQL INCLUDE statement can be used to provide the declaration of the SQLCA in all host languages except Java and REXX. For information on the use of the SQLCA in a REXX program, see Chapter 18, "Coding SQL statements in REXX applications," on page 1117. For information on how to access the information regarding errors and warnings in Java, see Chapter 17, "Coding SQL statements in Java applications," on page 1099.

In COBOL and C, the name of the storage area must be SQLCA. Every SQL statement must be within the scope of its declaration.

If stand-alone SQLCODE or SQLSTATE is used, an SQLCA cannot be included. For more information, see "SQL diagnostic information" on page 526.

The stand-alone SQLCODE and stand-alone SQLSTATE must not be specified in Java or REXX.

Field descriptions

Table 70. Field descriptions for an SQLCA

C Name[169]	COBOL Name	Field Data Type	Field Value
sqlcaid	SQLCAID	CHAR(8)	Contains an 'eye catcher' for storage dumps, 'SQLCA'.
sqlcabc	SQLCABC	INTEGER	Contains the length of the SQLCA, 136.
sqlcode	SQLCODE	INTEGER	Contains an SQL return code: **0** Successful execution, although SQLWARN indicators (see below) might have been set. **positive** Successful execution, but with a warning condition. **negative** Error condition.
sqlerrml[170]	SQLERRML	SMALLINT	Contains the length for SQLERRMC, in the range 0 through 70. If the length is 0, the value of SQLERRMC is not pertinent.
sqlerrmc[170]	SQLERRMC	VARCHAR (70)	Contains information that is substituted for variables in the descriptions of error conditions. See the product references for further information.

169. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

170. In C and COBOL, SQLERRM includes SQLERRML and SQLERRMC.

Table 70. Field descriptions for an SQLCA (continued)

C Name[169]	COBOL Name	Field Data Type	Field Value
sqlerrp	SQLERRP	CHAR(8)	Begins with a three-letter identifier indicating the product: DSN for DB2 for z/OS QSQ for DB2 for i SQL for DB2 for LUW If the SQLCODE indicates an error condition, then this field contains the name of the module that returned the error. See "CONNECT (type 1)" on page 598 for additional information.
sqlerrd	SQLERRD	Array	Contains six INTEGER variables that provide diagnostic information. On successful return from an SQL procedure, the first SQLERRD variable contains the return status value from the SQL procedure. The third SQLERRD variable shows the number of rows affected after INSERT, MERGE, UPDATE, and DELETE. For the TRUNCATE statement, the value is -1. If a PREPARE statement is successful, the fourth SQLERRD variable contains a relative cost estimate of the resources required to process the prepared statement. The fifth SQLERRD variable shows the number of rows affected by referential constraints as a result of a delete operation. G In DB2 for z/OS, the use of the fifth SQLERRD variable G is not supported.
sqlwarn	SQLWARN	Array	Contains a set of warning indicators. Each indicator is either blank or contains a value as indicated below.
SQLWARN0	SQLWARN0	CHAR(1)	Contains 'W' if at least one other indicator contains 'W'; it is blank if all the other indicators do not indicate a warning condition.
SQLWARN1	SQLWARN1	CHAR(1)	Contains 'W' if the value of a string column was truncated when assigned to a variable.
SQLWARN2	SQLWARN2	CHAR(1)	Contains 'W' if null values were eliminated from the argument of an aggregate function; not necessarily set to 'W' for the MIN function because its results are not dependent on the elimination of null values.
SQLWARN3	SQLWARN3	CHAR(1)	Contains 'W' if the number of columns is larger than the number of variables.
SQLWARN4	SQLWARN4	CHAR(1)	Contains 'W' if a prepared UPDATE or DELETE statement does not include a WHERE clause.
G SQLWARN5	SQLWARN5	CHAR(1)	Contents are product-specific.
SQLWARN6	SQLWARN6	CHAR(1)	Contains 'W' if date arithmetic results in an end of month adjustment. For more information, see "Incrementing and decrementing dates" on page 139.
G SQLWARN7	SQLWARN7	CHAR(1)	Contents are product-specific.

Table 70. Field descriptions for an SQLCA (continued)

C Name[169]	COBOL Name	Field Data Type	Field Value
SQLWARN8	SQLWARN8	CHAR(1)	Contains 'W' if a character that could not be converted was replaced with a substitution character.
G SQLWARN9	SQLWARN9	CHAR(1)	Contents are product-specific.
G SQLWARNA	SQLWARNA	CHAR(1)	Contents are product-specific.
sqlstate	SQLSTATE	CHAR(5)	A return code as described in Chapter 13, "SQLSTATE values—common return codes," on page 997. that indicates the outcome of the most recently executed SQL statement.

INCLUDE SQLCA declarations

For C

In C, INCLUDE SQLCA declarations are equivalent (but not necessarily identical) 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;
```

Figure 10. INCLUDE SQLCA declarations for C

For COBOL

In COBOL, INCLUDE SQLCA declarations are equivalent (but not necessarily identical) 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).
```

Figure 11. INCLUDE SQLCA declarations for COBOL

Chapter 12. SQLDA (SQL descriptor area)

An SQLDA is a set of variables that is required for execution of the SQL DESCRIBE statement, and it may optionally be used by the PREPARE, OPEN, FETCH, CALL, and EXECUTE statements. An SQLDA can be used in a DESCRIBE or PREPARE statement, altered with the addresses of storage areas[171], and then reused in a FETCH statement. It can also be used in OPEN, EXECUTE or CALL statements to provide input values or output variables.

SQLDAs are supported, with predefined declarations, for C, COBOL and REXX. In REXX, the SQLDA is somewhat different than in the other languages; for information on the use of SQLDAs in REXX, see "Defining SQL descriptor areas in REXX" on page 1118.

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, 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, 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 distinct types the base SQLVARs are followed by the same number of occurrences of an *extended SQLVAR*.

Base SQLVAR
The base SQLVAR entry is always present in an SQLDA. The fields of the base SQLVAR entry contain information about the column or storage area including data type, length attribute (except for LOBs), column name, CCSID, storage area address for data, and storage area address for an indicator.

Extended SQLVAR
The extended SQLVAR entry is used (for each column or variable) if the SQLDA includes any LOBs or distinct types. Each extended SQLVAR entry provides extended information for the corresponding base SQLVAR entry.

171. 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.

For distinct types, the extended SQLVAR contains the distinct type name. For LOBs, the extended SQLVAR contains the length attribute of the storage area and a pointer to the storage area that contains the actual length. If locators or file reference variables are used to represent LOBs, an extended SQLVAR is not necessary. If the corresponding base SQLVAR represents neither a LOB or distinct type, the extended SQLVAR includes no additional information.

Field descriptions in an SQLDA header

Table 71. Field descriptions for an SQLDA header

C Name[172, 173] **COBOL Name**	**Field Data Type**	**Usage in DESCRIBE or PREPARE (set by the database manager except for SQLN)**	**Usage in FETCH, OPEN, CALL, or EXECUTE (set by the application 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 occurrences of SQLVAR entries are needed" on page 987.	A '2' in the 7th byte indicates that two SQLVAR entries were allocated for each column. If the SQLNAME field contains an overriding CCSID, the 6th byte must be set to a '+' character.
sqldabc SQLDABC	INTEGER	Number of bytes of storage for 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*(N), where N is the length of an SQLVAR occurrence.[174]
sqln SQLN	SMALLINT	Before invoking DESCRIBE or PREPARE, set to the total number of occurrences of SQLVAR entries allocated for the SQLDA. The value is not changed by the database manager during DESCRIBE or PREPARE.	Total number of occurrences of SQLVAR entries allocated in the SQLDA. SQLN must be set to a value greater than or equal to SQLD. If LOBs types are included, extended SQLVARs are required. SQLN must be set to two times the number of parameter markers in the statement.
sqld SQLD	SMALLINT	The number of columns in the result table of the select-statement. Zero if the statement being described is not a select-statement.	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.

172. The field names shown are those present in an SQLDA that is obtained via an INCLUDE statement.

173. In this column, the lowercase name is the "C Name." The uppercase name is the "COBOL Name."

174. The value of N varies depending on the environment. For portability, the value should be calculated using an appropriate language sizing function. For example, in C use the sizeof() function to determine the size of the SQLVAR.

Determining how many occurrences of SQLVAR entries 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 Table 72 for more information.

If more than 1 set of SQLVARs is needed, the 7th byte of SQLDAID is set to the number of sets of SQLVARs necessary.

If SQLD is not set to a sufficient number of SQLVAR occurrences:

- SQLD is set to the total number of SQLVAR occurrences needed for all sets.
- 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.

Table 72 shows 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. 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 72. Contents of SQLVAR arrays

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)
No	No	Blank	n	Data type information	Not used
Yes	No	2	2n	Data type information with no length for LOB entries	LOB length for LOB entries
No	Yes	2	2n	Data type information except source data type information for distinct type entries	distinct type name
Yes	Yes	2	2n	Data type information with no length for LOB entries and source data type for distinct type entries	LOBs length for LOB entries and distinct type name for distinct type entries

Field descriptions in an occurrence of SQLVAR

Fields in an occurrence of a base SQLVAR

Table 73. Field descriptions for a base SQLVAR

C Name[172, 173] COBOL Name	Field Data Type	Usage in DESCRIBE or PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the user prior to executing the statement)
sqltype SQLTYPE	SMALLINT	The data type of the column and whether it can contain nulls. For a description of the type codes, see Table 75 on page 990. 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.	The data type of the host variable and whether an indicator variable is provided. For a description of the type codes, see Table 75 on page 990.
sqllen SQLLEN	SMALLINT	The length attribute of the column. For datetime columns, the length of the string representation of the values. See Table 75 on page 990. For a LOB, the value is 0 regardless of the length attribute of the LOB. 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 75 on page 990. 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.
sqldata SQLDATA	pointer[175]	For string columns or XML, the CCSID of the column. For datetime columns, SQLDATA can contain the CCSID of the string representation of the datetime value. See Table 76 on page 992 for the format of the field.	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.
sqlind SQLIND	pointer	Reserved	Contains the address of the indicator variable. Not used if there is no indicator variable (as indicated by an even value of SQLTYPE).

Table 73. Field descriptions for a base SQLVAR (continued)

C Name[172, 173] COBOL Name	Field Data Type	Usage in DESCRIBE or PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the user prior to executing the statement)
G G sqlname G SQLNAME G G G G	VARCHAR (30)	The unqualified name of the column. If the column does not have a name, the contents are product-specific. The name is case sensitive and does not contain surrounding delimiters.	For SQLVARs representing string types, the CCSID of the string. See Table 76 on page 992 for the format of the field. 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: Must be X'0000' . • Bytes 5 and 6 X'0100' XML host variable • Bytes 7 and 8: Must be X'0000'.

Fields in an occurrence of an extended SQLVAR

Table 74. Field descriptions for an extended SQLVAR

C Name[172, 173] COBOL Name	Field Data Type	Usage in DESCRIBE or PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the user prior to executing the statement)
len.sqllonglen SQLLONGLEN	INTEGER	The length attribute of a LOB column. For XML, the value is 0.	The length attribute of a LOB host variable. The length attribute indicates the number of bytes for a BLOB or CLOB, or XML and the number of double-byte characters for a DBCLOB. The database manager ignores the SQLLEN field in the base SQLVAR for these data types.

175. There may be additional reserved bytes preceding this field to properly align the pointer. See each product's SQLDA include file for details.

176. There are additional reserved bytes preceding this field to properly align the pointer and make the structure the same size as the base SQLVAR. See each product's SQLDA include file for details.

Table 74. Field descriptions for an extended SQLVAR (continued)

C Name[172, 173] COBOL Name	Field Data Type	Usage in DESCRIBE or PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the user prior to executing the statement)
sqldatalen SQLDATALEN	pointer[176]	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.
sqldatatype_name SQLDATATYPE-NAME	VARCHAR (30)	The fully qualified distinct type name for a distinct type column.	Not used.

SQLTYPE and SQLLEN

The following table shows the values that may appear in the SQLTYPE and SQLLEN fields of the SQLDA. In an SQLDA used in DESCRIBE or PREPARE statements, 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 null value.

In an SQLDA used in FETCH, OPEN, or EXECUTE statements, 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 75. SQLTYPE and SQLLEN values

	For DESCRIBE and PREPARE		For FETCH, OPEN, CALL, and EXECUTE	
SQLTYPE	**Column Data Type**	**SQLLEN**	**Host Variable Data Type**	**SQLLEN**
384/385	date	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

Table 75. SQLTYPE and SQLLEN values (continued)

	For DESCRIBE and PREPARE		For FETCH, OPEN, CALL, and EXECUTE	
SQLTYPE	**Column Data Type**	**SQLLEN**	**Host Variable Data Type**	**SQLLEN**
392/393	timestamp	26	fixed-length character string representation of a timestamp	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 [177]	BLOB	Not used. [177]
408/409	CLOB	0 [177]	CLOB	Not used. [177]
412/413	DBCLOB	0 [177]	DBCLOB	Not used. [177]
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
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 [178]	precision in byte 1; scale in byte 2	zoned decimal [178]	precision in byte 1; scale in byte 2
492/493	big integer	8	big integer	8
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2
504/505	Not Applicable	Not Applicable	DISPLAY SIGN LEADING SEPARATE [179]	precision in byte 1; scale in byte 2
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
996/997	DECFLOAT(16) DECFLOAT(34)	8 16	DECFLOAT(16) DECFLOAT(34)	8 16

Table 75. SQLTYPE and SQLLEN values (continued)

	For DESCRIBE and PREPARE		For FETCH, OPEN, CALL, and EXECUTE	
SQLTYPE	**Column Data Type**	**SQLLEN**	**Host Variable Data Type**	**SQLLEN**
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

CCSID values in SQLDATA and 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 following must be true:

- the sixth byte of the SQLDAID in the SQLDA header is set to '+'
- the SQLNAME length is set to 8
- the first 4 bytes of SQLNAME are set as described in the Table 76.

In the DESCRIBE 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. If the column is a datetime column, the SQLDATA field of the SQLVAR can contain the CCSID of the string representation of the datetime value. The CCSID is located in bytes 3 and 4 as described in Table 76.

Table 76. CCSID values for SQLDATA[1] or SQLNAME

Data Type	Subtype	Bytes 1 & 2	Bytes 3 & 4
Character	SBCS data	X'0000'	ccsid
Character	Mixed data	X'0000'	ccsid
Character	Bit data	X'0000'	X'FFFF'[2]
Graphic	Not Applicable	X'0000'	ccsid
Datetime	Not Applicable	X'0000'	ccsid
Any other data type	Not Applicable	Not Applicable	Not Applicable

Notes:

1. In DB2 for LUW, the value for SQLDATA does not follow this format on a 64-bit systems or systems using little endian integer formats. In these cases, the CCSID value can be returned by casting the value as an integer.
2. In DB2 for LUW, X'0000' is returned instead of X'FFFF' for bit data.

177. Field SQLLONGLEN in the extended SQLVAR contains the length attribute of the column.

178. In DB2 for z/OS, and DB2 for LUW, zoned decimal is not supported for local operations.

179. In DB2 for LUW, DISPLAY SIGN LEADING SEPARATE is not supported.

180. DB2 for i supports DECFLOAT(7) numbers. It 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

For C

In C, INCLUDE SQLDA declarations are equivalent (but not necessarily identical) to the following:

```
#ifndef SQLDASIZE
struct sqlda
{
       unsigned char  sqldaid[8];
       long           sqldabc;
       short          sqln;
       short          sqld;
       struct sqlvar
       {
              short          sqltype;
              short          sqllen;
              unsigned char *sqldata;
              short         *sqlind;
              struct sqlname
              {
                     short          length;
                     unsigned char  data[30];
              } sqlname;
       } sqlvar[1];
};
struct sqlvar2
       { struct
                {  long           sqllonglen;
                   char           reserve1[SQLVAR2_PAD];
                }  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 12. INCLUDE SQLDA declarations for C

```
/*****************************************************************/
/* Macros for using the sqlvar2 fields.                          */
/*****************************************************************/

/*****************************************************************/
/*   '2' in the 7th byte of sqldaid indicates a doubled 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

In COBOL, INCLUDE SQLDA declarations are equivalent[181] (but not necessarily identical) 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 SQLIND    POINTER.
        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         REDEFINES 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 13. INCLUDE SQLDA declarations for COBOL

181. The line starting with SQLVAR OCCURS has a different value in the include for each platform with 409 repsenting the lowest value. If this value is too low, a portable application should code the SQLDA definition directly, specifying the value required by the application.

Chapter 13. SQLSTATE values—common return codes

This appendix contains a summary of return codes called SQLSTATE values that are defined for the DB2 SQL relational database products. SQLSTATE values are produced when an SQL statement is executed. The SQLSTATE values provide application programs with common return codes for common error conditions. Return codes from other database operations (such as commands) are not included.

This summary includes SQLSTATE values that cover existing conditions from all of the DB2 SQL relational database products. Many of these conditions are product-specific. These values have been included for the convenience of application developers concerned with a distributed database environment where any of these values could be returned.

The SQLSTATE values are consistent with the SQLSTATE specifications contained in SQL 2011 Core standard.

Using SQLSTATE values

An SQLSTATE value is a return code that indicates the outcome of the most recently executed SQL statement. The mechanism used to access SQLSTATE values depends on where the SQL statement is executed:

- In embedded applications other than Java, SQLSTATE values are returned in the last five bytes of the SQLCA or in a stand-alone SQLSTATE variable. For more information see, "SQL diagnostic information" on page 526.
- In Java, SQLSTATE values are returned by using getSQLState() method. For more information see, Chapter 17, "Coding SQL statements in Java applications," on page 1099.

SQLSTATE values are designed so that application programs can test for specific conditions or classes of conditions.

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. Programmers who want to use SQLSTATE as the basis of their applications' return codes can define their own SQLSTATE classes or subclasses:

- 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.

The class code of an SQLSTATE value indicates whether the SQL statement was executed successfully (class codes 00 and 01) or unsuccessfully (all other class codes).

Table 1 identifies the SQLSTATE class codes used by DB2 SQL and the SQL 2011 Core standard.

Table 1. SQLSTATE Class Codes

Class Code	Meaning	Subclass Code Table
00	Unqualified Successful Completion	Table 2
01	Warning	Table 3
02	No Data	Table 4
03	SQL Statement Not Yet Complete	Table 5
07	Dynamic SQL Error	Table 6
08	Connection Exception	Table 7
09	Triggered Action Exception	Table 8
0A	Feature Not Supported	Table 9
0D	Invalid Target Type Specification	Table 10
0E	Invalid Schema Name List Specification	Table 11
0F	Invalid Token	Table 12
0K	Resignal When Handler Not Active	Table 13
0N	SQL/XML Mapping Error	Table 14
0W	Prohibited Statement Encountered During Trigger	Table 15
0Z	Diagnostics Exception	Table 16
10	XQuery Error	Table 17
20	Case Not Found for Case Statement	Table 18
21	Cardinality Violation	Table 19
22	Data Exception	Table 20
23	Constraint Violation	Table 21
24	Invalid Cursor State	Table 22
25	Invalid Transaction State	Table 23
26	Invalid SQL Statement Identifier	Table 24
27	Triggered Data Change Violation	Table 25
28	Invalid Authorization Specification	Table 26
2D	Invalid Transaction Termination	Table 27
2E	Invalid Connection Name	Table 28
2F	SQL Function Exception	Table 29
33	Invalid SQL Descriptor Name	Table 30
34	Invalid Cursor Name	Table 31
35	Invalid Condition Number	Table 32
36	Cursor Sensitivity Exception	Table 33
38	External Function Exception	Table 34
39	External Function Call Exception	Table 35
3B	Savepoint Exception	Table 36
3C	Ambiguous Cursor Name	Table 37
3F	Invalid Schema Name	Table 38
40	Transaction Rollback	Table 39
42	Syntax Error or Access Rule Violation	Table 40
44	WITH CHECK OPTION Violation	Table 41
46	Java Errors	Table 42

Class Code	Meaning	Subclass Code Table
51	Invalid Application State	Table 43
53	Invalid Operand or Inconsistent Specification	Table 44
54	SQL or Product Limit Exceeded	Table 45
55	Object Not in Prerequisite State	Table 46
56	Miscellaneous SQL or Product Error	Table 47
57	Resource Not Available or Operator Intervention	Table 48
58	System Error	Table 49
5U	Common Utilities and Tools	Table 50

Table 2. Class Code 00: Unqualified Successful Completion

SQLSTATE	Meaning
00000	Execution of the SQL statement was successful and did not result in any type of warning or exception condition.

Table 3. Class Code 01: Warning

SQLSTATE Value	Meaning
01xxx	Valid warning SQLSTATEs returned by an SQL routine. Also used for RAISE_ERROR and SIGNAL.
01002	A DISCONNECT error occurred.
01003	Null values were eliminated from the argument of an aggregate function.
01004	The value of a string was truncated when assigned to another string data type with a shorter length.
01005	Insufficient number of entries in an SQLDA.
01006	A privilege was not revoked.
01007	A privilege was not granted.
01009	The search condition is too long for the information schema.
0100A	The query expression of the view is too long for the information schema.
0100C	One or more ad hoc result sets were returned from the procedure.
0100D	The cursor that was closed has been reopened on the next result set within the chain.
0100E	The procedure returned too many result sets.
01011	The PATH value has been truncated. Array data, right truncation.
01503	The number of result columns is larger than the number of variables provided.
01504	The UPDATE or DELETE statement does not include a WHERE clause.
01505	The statement was not executed because it is unacceptable in this environment.
01506	An adjustment was made to a DATE or TIMESTAMP value to correct an invalid date resulting from an arithmetic operation.
01507	One or more non-zero digits were eliminated from the fractional part of a number used as the operand of a multiply or divide operation.
01508	The statement was disqualified for blocking for reasons other than storage.
01509	Blocking was cancelled for a cursor because there is insufficient storage in the user virtual machine.

SQLSTATE Value	Meaning
01510	Blocking was cancelled for a cursor because a blocking factor of at least two rows could not be maintained.
01511	Performance may not be optimum because of the number of predicates specified in the WHERE clause.
01512	The REVOKE operation has no effect on CONNECT privileges.
01513	A subsequent commit operation will revoke all EXECUTE privileges on the package except for that of the owner.
01514	The tablespace has been placed in the check-pending state.
01515	The null value has been assigned to a variable, because the non-null value of the column is not within the range of the variable.
01516	An inapplicable WITH GRANT OPTION has been ignored.
01517	A character that could not be converted was replaced with a substitute character.
01519	The null value has been assigned to a variable, because a numeric value is out of range.
01520	The null value has been assigned to a variable, because the characters cannot be converted.
01521	A specified server-name is undefined but is not needed until the statement is executed or the alias is used.
01522	The local table or view name used in the CREATE ALIAS statement is undefined.
01523	ALL was interpreted to exclude ALTER, INDEX, REFERENCES, and TRIGGER, because these privileges cannot be granted to a remote user.
01524	The result of an aggregate function does not include the null values that were caused by evaluating the arithmetic expression implied by the column of the view.
01525	The number of INSERT values is not the same as the number of columns.
01526	Isolation level has been escalated.
01527	A SET statement references a special register that does not exist at the AS.
01528	WHERE NOT NULL is ignored, because the index key cannot contain null values.
01530	Definition change may require a corresponding change on the read-only systems.
01532	An undefined object name was detected.
01533	An undefined column name was detected.
01534	The string representation of a datetime value is invalid.
01535	An arithmetic operation on a date or timestamp has a result that is not within the valid range of dates.
01536	During remote bind where existence checking is deferred, the server-name specified does not match the current server.
01537	An SQL statement cannot be EXPLAINed, because it references a remote object.
01538	The table cannot be subsequently defined as a dependent, because it has the maximum number of columns.
01539	Connection is successful but only SBCS characters should be used.
01540	A limit key has been truncated to 40 bytes.
01541	Operator command processing has completed successfully.
01542	Authorization ID does not have the privilege to perform the operation as specified.
01543	A duplicate constraint has been ignored.
01544	The null value has been assigned to a variable, because a substring error occurred; for example, an argument of SUBSTR is out of range.

SQLSTATE Value	Meaning
01545	An unqualified column name has been interpreted as a correlated reference.
01546	A column of the explanation table is improperly defined.
01547	A mixed data value is improperly formed.
01548	The authorization ID does not have the privilege to perform the specified operation on the identified object.
01550	The object was not created, because an object with the specified description already exists.
01551	A table in a partitioned tablespace is not available, because its partitioned index has not been created.
01552	An ambiguous qualified column name was resolved to the first of the duplicate names in the FROM clause.
01553	Isolation level RR conflicts with a tablespace locksize of page.
01554	Decimal multiplication may cause overflow.
01555	Mixed data is invalid and has been truncated according to SBCS rules.
01557	Too many variables have been specified on SELECT INTO or FETCH.
01558	A distribution protocol has been violated.
01560	A redundant GRANT has been ignored.
01561	An update to a data capture table was not signaled to the originating subsystem.
01562	The new path to the log (newlogpath) in the database configuration file is invalid.
01563	The current path to the log file (logpath) is invalid. The log file path is reset to the default.
01564	The null value has been assigned to a variable, because division by zero occurred.
01565	The null value has been assigned to a variable, because a miscellaneous data exception occurred. For example, the character value for the CAST, DECIMAL, FLOAT, or INTEGER scalar function is invalid; a floating-point NAN (not a number); invalid data in a packed decimal field; or a mask mapping error was detected.
01566	The index has been placed in a pending state.
01567	The table was created but not journaled.
01568	The dynamic SQL statement ends with a semicolon.
01570	The bind process detected a character string in an INSERT or UPDATE statement that is too large for the target column.
01571	The bind process detected a numeric value that is out of range.
01572	The bind process detected an invalid datetime format, such as an invalid string representation or an invalid value.
01573	The bind process detected a null insert or update value that is null for a column that cannot contain null values.
01574	The bind process detected an INSERT, UPDATE, or DELETE that is not permitted on this object.
01575	The bind process detected a non-updatable column in an INSERT or UPDATE statement.
01576	The bind process detected a CREATE INDEX statement for a view.
01577	The bind process detected a CREATE VIEW statement that includes an operator or operand that is not valid for views.
01578	The bind process detected operands of an operator that are not compatible.
01579	The bind process detected a numeric constant that is either too long or has a value that is not within the range of its data type.
01580	The bind process detected an update or insert value that is not compatible with the column.

SQLSTATE Value	Meaning
01581	The bind process detected incompatible operands of a UNION operator.
01582	The bind process detected a string that is too long.
01583	The bind process detected a decimal divide operation that is invalid, because the result would have a negative scale.
01584	The bind process detected an insert or update value of a long string column that is neither a variable nor NULL.
01585	The bind process detected a table that cannot be accessed, because it is inactive.
01586	Processing the statement resulted in one or more tables being automatically placed into a set-integrity-pending state.
01587	The unit of work was committed or rolled back, but the outcome is not fully known at all sites.
01588	The LIKE predicate has an invalid escape character.
01589	A statement contains redundant specifications.
01590	Type 2 indexes do not have subpages.
01591	The result of the positioned UPDATE or DELETE may depend on the order of the rows.
01593	An ALTER TABLE may cause data truncation.
01594	Insufficient number of entries in an SQLDA for ALL information (i.e. not enough descriptors to return the distinct name).
01595	The view has replaced an existing, invalidated view.
01596	Comparison functions were not created for a distinct type based on a long string data type.
01597	Specific and non-specific volume IDs are not allowed in a storage group.
01598	An attempt has been made to set the state of an event monitor or usage list to its current state.
01599	Bind options were ignored on REBIND.
01600	SUBPAGES ignored on alter of catalog index.
01602	Optimization processing encountered a restriction that might have caused it to produce a sub-optimal result.
01603	CHECK DATA processing found constraint violations and moved them to exception tables.
01604	The SQL statement was explained and not executed.
01605	A recursive common table expression may contain an infinite loop.
01606	The node or system database directory is empty.
01607	The difference in the times on members in a read-only transaction exceed the defined threshold.
01608	An unsupported value has been replaced.
01614	There are fewer locators than the number of result sets.
01616	The estimated CPU cost exceeds the resource limit.
01618	The database partitioning must be changed by redistributing the database partition group.
01620	Some base tables of UNION ALL may be the same table.
01621	The retrieved LOB value may have been changed.
01622	Statement completed successfully but a system error occurred after the statement completed."
01623	The value of DEGREE is ignored.
01624	The GBPCACHE specification is ignored because the buffer pool does not allow caching.
01625	The schema name appears more than once in the CURRENT PATH.
01626	The database has only one buffer pool.

SQLSTATE Value	Meaning
01627	The DATALINK value may not be valid because the table is in reconcile pending or reconcile is not a possible state.
01628	The user-specified access path hints are invalid. The access path hints are ignored.
01629	User-specified access path hints were used during access path selection.
01632	The number of concurrent connections has exceeded the defined entitlement for the product.
01633	The materialized query table may not be used to optimize the processing of queries.
01634	The distinct data type name is too long and cannot be returned in the SQLDA. The short name is returned instead.
01636	Integrity of non-incremental data remains unverified by the database manager.
01637	Debugging is not enabled.
01639	The federated object may require the invoker to have necessary privileges on data source objects.
01640	ROLLBACK TO SAVEPOINT occurred when there were uncommitted INSERTs or DELETEs that cannot be rolled back.
01642	Column not long enough for the largest possible USER default value.
01643	Assignment to SQLCODE or SQLSTATE variable does not signal a warning or error.
01644	DEFINE NO is not applicable for a lob space or data sets using the VCAT option.
01645	The executable for the SQL procedure is not saved in the catalog.
01646	A result sets could not be returned because the cursor was closed.
01647	A DB2SQL BEFORE trigger changed to DB2ROW.
01648	COMPRESS column attribute ignored because VALUE COMPRESSION has not been activated for the table.
01649	The buffer pool configuration has been completed but will not take effect until the next database restart.
01650	Index and table statistics are inconsistent.
01651	The event monitor was activated successfully, however some monitoring information may be lost.
01652	The isolation clause was ignored because of the statement context.
01653	The authorizations were granted to the user, but groups were not considered since the authorization name is more than 8 bytes.
01654	The buffer pool is not started.
01655	The event monitor was created successfully but at least one event monitor target table already exists.
01656	ROLLBACK TO savepoint caused a NOT LOGGED table space to be placed in the LPL.
01657	The buffer pool operation will not take effect until the next database restart.
01658	Binary data is invalid for DECRYPT_CHAR and DECYRYPT_DB.
01659	A non-atomic statement successfully processed all requested rows with one or more warning conditions.
01660	The routine was created but a restore will not update the catalog.
01661	A violation of the constraint imposed by a unique index or a unique constraint occurred.
01662	Release record option ignored on CLOSE.
01663	NOT PADDED clause is ignored for indexes created on auxiliary tables.
01664	Option not specified following the ALTER PARTITION CLAUSE.
01665	A name or label was truncated.

SQLSTATE Value	Meaning
01666	The last partition's limit key value is set to the highest or lowest possible value.
01667	The view may not be used to optimize the processing of queries.
01668	A rowset FETCH statement returned one or more rows of data, with one or more bind out processing error conditions. Use GET DIAGNOSTICS for more information.
01669	The statistics for the specified nicknames were not completely updated because of schema inconsistencies between the remote and local catalogs.
01670	No default primary table space exists for the new table.
01671	The compilation environment of the specified cached statement is different than the current compilation environment.
01672	The alter table statement was a REORG recommended alter. Use the REORG utility to reorganize the table.
01673	All roles, including the role specified, are already enabled.
01674	Table space attributes are not optimal for query performance.
01675	More table spaces than required were specified. The extra table spaces are ignored.
01676	Transfer operation ignored since the authorization ID is already the owner of the database object.
01677	Wrapper options were ignored for servers that already have the plugin defined.
01678	Changes to the user mapping apply only to the federated catalog table and not to the external user mapping repository.
01679	A trusted connection cannot be established for the specified system authorization ID.
01680	The option is not supported in the context in which it was specified.
01681	The trusted context is no longer defined to be used by specific attribute value.
01682	The ability to use the trusted context was removed from some but not all authorization IDs specified in statement.
01683	A SELECT containing a non-ATOMIC data change statement successfully returned some rows, but one or more warnings or errors occurred.
01684	he specified locale is not supported. The message was returned in the English locale.
01685	An invalid use of a NOT DETERMINISTIC or EXTERNAL ACTION function was detected.
01687	A database resource was not available. Processing continues.
01689	The SQL compilation completed without connecting to the data source.
0168A	The package body for the source procedure at the data source was not found or is invalid.
0168B	An operation was partially successful and partially unsuccessful. Use GET DIAGNOSTICS for more information.
0168C	A decimal float operation produced an inexact result.
0168D	A decimal floating point operation was invalid.
0168E	A decimal float operation produced an overflow or underflow.
0168F	A decimal float operation produced division by zero.
0168G	A decimal float operation produced a subnormal number.
0168H	The product is running in evaluation mode. A valid license key is not installed.
0168I	The SQL statement does conform to the specified flagging level.
0168J	The table space could not be reduced in size.
0168L	No routine was found with the specified name and compatible arguments.
0168M	Changing the database configuration parameter DECFLT_ROUNDING may have unintended consequences.

SQLSTATE Value	Meaning
0168O	The federated server received an unknown warning from a data source.
0168P	An associated mixed or graphic CCSID does not exist for the default job CCSID.
0168Q	The wrapper supports current data source server versions as listed. Using the wrappers with later versions might result in errors or unexpected results.
0168R	The text index may be out of date.
0168S	A task was not removed.
0168T	WITH ROW CHANGE COLUMNS ALWAYS DISTINCT was specified, but the database manager is unable to return distinct row change columns.
0168U	Result sets will not be returned to the client because the procedure was called directly or indirectly from a function or trigger.
0168V	SYSTEM SAMPLING was specified for RUNSTATS but is not supported for the statistical view specified. BERNOLLI SAMPLING was done instead.
0168W	The database partition has been added successfully but will not be operational until the instance is restarted.
0168X	The combination of target namespace and schema location hint is not unique in the XML schema repository.
0168Y	The newly defined object is marked invalid because it references an object which is not defined, or is invalid, or the definer does not have privilege to access it.
0168Z	The statement was successfully prepared, but cannot be executed.
01690	The rebalance operation did not need to move any data, or data is being moved but not all stripe sets have a container on each storage path.
01691	The storage path was not dropped but is in the drop pending state because one or more automatic storage table spaces reside on the path.
01693	PROGRAM TYPE SUB changed to PROGRAM TYPE MAIN.
01694	A deprecated feature has been ignored.
01695	Adjustment made to a value for a period as a result of a data change operation.
01696	One or more tables in the schema have a different attribute than the schema.
01698	Permissions or masks of a materialized query table might require changes as a result of changes to permissions or masks of the table on which it is based.
01699	An in-database analytics provider returned an unexpected warning.
0169A	A configuration parameter was overridden.
0169B	The operation was successful on the DB2 server, but may not have been successful on the accelerator server.
01Hxx	Valid warning SQLSTATEs returned by a user-defined function, external procedure CALL, or command invocation.
01H51	An MQSeries Application Messaging Interface message was truncated.
01H52	Routine execution has completed, but at least one error or warning was encountered during the execution. More information is available.
01H53	The routine has encountered a warning. Refer to the SQLCODE for details.
01H54	The procedure has returned successfully but encountered an error in the format or content of a parameter. Information about the error in the parameter value is returned in an output parameter.
01H55	The procedure has returned successfully but encountered an internal processing error. Information about the internal error situation is returned in an output parameter.
01H56	The procedure has returned successfully but supports a higher version for a parameter than the one that was specified.

SQLSTATE Value	Meaning
01H57	The procedure has returned output in an alternate locale instead of the locale specified.
01HN0	An enabled workload is associated with a disabled service class.
01HN1	A priority setting for a service class was set higher than the priority setting for the default system service class SYSDEFAULTSYSTEMCLASS and this may negatively impact performance.
01HN2	LOAD HADOOP statement completed with information.

Table 4. Class Code 02: No Data

SQLSTATE	Meaning
02000	One of the following exceptions occurred: • The result of the SELECT INTO statement or the subselect of the INSERT statement was an empty table. • The number of rows identified in the searched UPDATE or DELETE statement was zero. • The position of the cursor referenced in the FETCH statement was after the last row of the result table. • The fetch orientation is invalid.
02001	No additional result sets returned.
02502	Delete or update hole detected.
02503	The procedure identified in an ALLOCATE CURSOR statement did not return hoc result sets.
02504	FETCH PRIOR ROWSET returned a partial rowset.
02505	The GET DESCRIPTOR VALUE is greater than COUNT.
02506	Errors were encountered and tolerated as specified by the RETURN DATA UNTIL clause or the RETURN EMPTY FOR clause.

Table 5. Class Code 03: SQL Statement Not Yet Complete

SQLSTATE	Meaning
03000	Asynchronous execution is not yet completed.

Table 6. Class Code 07: Dynamic SQL Error

SQLSTATE	Meaning
07001	The number of variables is not correct for the number of parameter markers.
07002	The call parameter list or control block is invalid.
07003	The statement identified in the EXECUTE statement is a select-statement, or is not in a prepared state.
07004	The USING clause or INTO clause is required for dynamic parameters.
07005	The statement name of the cursor identifies a prepared statement that cannot be associated with a cursor.
07006	An input variable, transition variable, or parameter marker cannot be used, because of its data type.
07007	The dynamic statement requires a result area and none was specified.
07008	The descriptor count is invalid.
07009	The descriptor index is invalid.

SQLSTATE	Meaning
0700C	Undefined DATA value.
0700E	Invalid LEVEL specified in SET DESCRIPTOR statement.
0700F	Invalid DATETIME_INTERVAL_CODE specified in SET DESCRIPTOR statement.
07501	The option specified on PREPARE is not valid.

Table 7. Class Code 08: Connection Exception

SQLSTATE	Meaning
08001	The application requester is unable to establish the connection.
08002	The connection already exists.
08003	The connection does not exist.
08004	The application server rejected establishment of the connection.
08007	Transaction resolution unknown.
08501	A DISCONNECT is not allowed when the connection uses an LU6.2 protected conversation.
08502	The CONNECT statement issued by an application process running with a SYNCPOINT of TWOPHASE has failed, because no transaction manager is available.
08504	An error was encountered while processing the path rename configuration file.
08505	Initialization of the continuous availability environment failed.
08506	A connection failed but has been re-established.
08507	A connection to the server failed to be re-established because of mismatched release levels.
08508	The remote host was not found.

Table 8. Class Code 09: Triggered Action Exception

SQLSTATE	Meaning
09000	A triggered SQL statement failed.

Table 9. Class Code 0A: Feature Not Supported

SQLSTATE	Meaning
0A001	The CONNECT statement is invalid, because the process is not in the connectable state.
0A502	The action or operation is not enabled for this database instance.
0A503	Federated insert, update, or delete operation cannot be compiled because of potential data inconsistency.
0A504	The driver does not support an optional feature.

Table 10. Class Code 0D: Invalid Target Type Specification

SQLSTATE	Meaning
0D000	The target structured data type specification is a proper subtype of the source structured data type.

Table 11. Class Code 0E: Invalid Schema Name List Specification

SQLSTATE	Meaning
0E000	The path name list is not valid.

Table 12. Class Code 0F: Invalid Token

SQLSTATE	Meaning
0F001	The locator value does not currently represent any value.

Table 13. Class Code 0K: Resignal When Handler Not Active

SQLSTATE	Meaning
0K000	A RESIGNAL was issued but a handler is not active.

Table 14. Class Code 0N: SQL/XML Mapping Error

SQLSTATE	Meaning
0N002	A character cannot be mapped to a valid XML character.

Table 15. Class Code 0W: Prohibited Statement Encountered During Trigger

SQLSTATE	Meaning
0W000	The statement is not allowed in a trigger.

Table 16. Class Code 0Z: Diagnostics Exception

SQLSTATE	Meaning
0Z001	Maximum number of stacked diagnostics areas exceeded.
0Z002	Stacked diagnostics accessed without and active handler.

Table 17. Class Code 10: XQuery Error

SQLSTATE	Meaning
10501	An XQuery expression is missing the assignment of a static or dynamic context component.
10502	An error was encountered in the prolog of an XQuery expression.
10503	A duplicate name was defined in an XQuery or XPath expression.
10504	An XQuery namespace declaration specified an invalid URI.
10505	A character, token or clause is missing or invalid in an XQuery expression.
10506	An XQuery expression references a name that is not defined.
10507	A type error was encountered processing an XPath or XQuery expression.
10508	An XQuery expression includes an invalid name expression or content expression.
10509	An unsupported XQuery language feature is specified.
10510	A string literal is not specified as the operand of a cast expression or as the argument of a constructor function.
10601	An arithmetic error was encountered processing an XQuery function or operator.
10602	A casting error was encountered processing an XQuery function or operator.

SQLSTATE	Meaning
10603	A character handling error was encountered processing an XQuery function or operator.
10605	A datetime error was encountered processing an XQuery function or operator.
10606	There is no context item for processing an XQuery function or operator.
10607	A namespace error was encountered processing an XQuery function or operator.
10608	An error was encountered in the argument of an XQuery function or operator.
10609	A regular expression error was encountered processing an XQuery function or operator.
10610	A type error was encountered processing an XQuery function or operator.
10611	An unidentified error was encountered processing an XQuery function or operator.
10701	An XQuery updating expression is used outside of the modify clause of a transform expression.or function.
10702	An XQuery expression in the modify clause of a transform expression is not an updating expression or an empty sequence.
10703	The target node of an XQuery basic updating expression is not valid.
10704	An XQuery transform expression includes incompatible basic updating expressions.
10705	An XQuery transform expression includes an assigned value in the copy clause that is not a single node.
10706	The replacement sequence of an XQuery replace expression contains invalid nodes.
10707	The result of an XQuery transform expression is not a valid instance of the XQuery and XPath data model.
10708	An XQuery updating expression introduces a new namespace binding that conflicts with another updating expression or the in-scope namespaces of an element node.
10709	A rename of a processing instruction node specified a QName with a prefix that is not empty.
10901	The length of a QName in an XQuery expression exceeds a product limit.
10902	An XQuery atomic value exceeds the length limit for a DB2 XQuery operator or function.
10903	An internal limit has been exceeded for the number of matched XQuery nodes.

Table 18. Class Code 20: Case Not Found for Case Statement

SQLSTATE	Meaning
20000	The case was not found for the CASE statement.

Table 19. Class Code 21: Cardinality Violation

SQLSTATE	Meaning
21000	The result of a SELECT INTO, scalar fullselect, or subquery of a basic predicate is more than one value.
21501	A multiple-row INSERT into a self-referencing table is invalid.
21502	A multiple-row UPDATE of a primary key is invalid.
21504	A multiple-row DELETE from a self-referencing table with a delete rule of RESTRICT or SET NULL is invalid.
21505	A row function must return not more than one row.
21506	The same row of the target table was identified more than once for an update, delete, or insert operation of the MERGE statement.

SQLSTATE	Meaning
21507	The result of the SQL statement specified for the administrative task results in more than one row or the wrong number of columns.

Table 20. Class Code 22: Data Exception

SQLSTATE	Meaning
22001	Character data, right truncation occurred; for example, an update or insert value is a string that is too long for the column, or a datetime value cannot be assigned to a variable, because it is too small.
22002	A null value, or the absence of an indicator parameter was detected; for example, the null value cannot be assigned to a variable, because no indicator variable is specified.
22003	A numeric value is out of range.
22004	A null value is not allowed.
22005	An error occurred on assignment.
22006	The fetch orientation is invalid.
22007	An invalid datetime format was detected; that is, an invalid string representation or value was specified.
22008	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates.
2200E	Null value in array target.
2200G	The most specific type does not match.
2200L	The XML value is not a well-formed document with a single root element.
2200M	The XML document is not valid.
2200S	The XML comment is not valid.
2200T	The XML processing instruction is not valid.
2200V	A context item is an XML sequence of more than one item.
2200W	An XML value contained data that could not be serialized.
22010	Invalid indicator parameter value.
22011	A substring error occurred; for example, an argument of SUBSTR or SUBSTRING is out of range.
22012	Division by zero is invalid.
22018	The character value for the CAST, DECIMAL, FLOAT, or INTEGER scalar function is invalid.
22019	The LIKE predicate has an invalid escape character.
2201G	Invalid argument for width bucket function.
2201R	The XML document is not valid.
2201S	Invalid XQuery regular expression.
2201T	Invalid option flag associated with regular expression.
2201V	Invalid replacement string for matched regular expression.
2201W	The row count in the FETCH FIRST clause or LIMIT clause is invalid.
2201X	The row count in the result OFFSET clause is invalid.
22021	A character is not in the coded character set or the conversion is not supported.
22023	A parameter or variable value is invalid.
22024	A NUL-terminated input host variable or parameter did not contain a NUL.
22025	The LIKE predicate string pattern contains an invalid occurrence of an escape character.

SQLSTATE	Meaning
2202D	A null instance was used with a mutator method.
2202E	Array element error.
2202F	Array data, right truncation.
2202H	The sample size in the TABLESAMPLE clause is invalid.
22501	The length control field of a variable length string is negative or greater than the maximum.
22502	Signalling NaN was encountered.
22503	The string representation of a name is invalid.
22504	A mixed data value is invalid.
22505	The local date or time length has been increased, but the executing program relies on the old length.
22506	A reference to a datetime special register is invalid, because the clock is malfunctioning or the operating system time zone parameter is out of range.
22508	CURRENT PACKAGESET is blank.
22511	ADT length exceeds maximum column length. The value for a ROWID or reference column is not valid.
22512	A variable in a predicate is invalid, because its indicator variable is negative.
22519	The primary or foreign key cannot be activated.
22521	The foreign key cannot be defined, because the primary key of the parent table is inactive.
22522	A CCSID value is not valid at all, not valid for the data type or subtype, or not valid for the encoding scheme.
22524	Character conversion resulted in truncation
22525	Partitioning key value is not valid.
22526	A key transform function generated no rows or duplicate rows.
22527	Invalid input data detected for a multiple-row insert.
22528	Binary data is invalid for DECRYPT_CHAR and DECYRYPT_DB.
22529	A non-atomic statement successfully completed for at least one row, but one or more errors occurred.
22530	A non-atomic statement attempted to process multiple rows of data, but no row was inserted and one or more errors occurred.
22531	The argument of a built-in or system provided routine resulted in an error.
22532	An XSROBJECT is not found in the XML schema repository.
22533	A unique XSROBJECT could not be found in the XML schema repository.
22534	An XML schema document is not connected to the other XML schema documents using an include or redefine.
22535	The XML schema does not declare the specified global element.
22536	The XML value does not contain the required root element.
22537	A rowset FETCH statement returned one or more rows of data, with one or more non-terminating error conditions. Use GET DIAGNOSTICS for more information.
22538	The XML schema update is not compatible with the existing XML schema.
22539	Invalid use of extended indicator parameter value.
22540	An UPDATE statement cannot have all columns set to UNASSIGNED.
22541	The binary XML value contains unrecognized data.
22542	The INSERT or UPDATE in not allowed because a resulting row does not satisfy row permissions.

SQLSTATE	Meaning
22544	The binary XML value contains a version that is not supported.
22545	Constructing an associative array failed because the input data includes duplicate array index values.
22546	The value for a routine argument is not valid.
22547	Multiple result values cannot be returned from the scalar function.
225D1	The specified XML schema is not enabled for decomposition.
225D2	An SQL Error occurred during decomposition of an XML document.
225D3	Decomposition of XML document encountered a value that is not valid for the XML schema type.
225D4	Decomposition of XML document encountered a value that is not valid for the target SQL type.
225D5	Decomposition of XML document encountered an XML node that is unknown or not valid in context.
225D6	The specified XML schema requires migration to current version to support decomposition.
225D7	Decomposition of XML document enountered root element that is not a global element of complexType in the XML schema.
225DE	An XML schema cannot be enabled for decomposition.
225X0	XSLT processor error.

Table 21. Class Code 23: Constraint Violation

SQLSTATE	Meaning
23001	The update or delete of a parent key is prevented by a RESTRICT update or delete rule.
23502	An insert or update value is null, but the column cannot contain null values.
23503	The insert or update value of a foreign key is invalid.
23504	The update or delete of a parent key is prevented by a NO ACTION update or delete rule.
23505	A violation of the constraint imposed by a unique index or a unique constraint occurred.
23506	A violation of a constraint imposed by an edit or validation procedure occurred.
23507	A violation of a constraint imposed by a field procedure occurred.
23508	A violation of a constraint imposed by the DDL Registration Facility occurred.
23509	The owner of the package has constrained its use to environments which do not include that of the application process.
23510	A violation of a constraint on the use of the command imposed by the RLST table occurred.
23511	A parent row cannot be deleted, because the check constraint restricts the deletion.
23512	The check constraint cannot be added, because the table contains rows that do not satisfy the constraint definition.
23513	The resulting row of the INSERT or UPDATE does not conform to the check constraint definition.
23514	Check data processing has found constraint violations.
23515	The unique index could not be created or unique constraint added, because the table contains duplicate values of the specified key.
23520	The foreign key cannot be defined, because all of its values are not equal to a parent key of the parent table.
23521	The update of a catalog table violates an internal constraint.
23522	The range of values for the identity column or sequence is exhausted.
23523	An invalid value has been provided for the SECURITY LABEL column.

SQLSTATE	Meaning
23524	Invalid row movement within the UNION ALL view.
23525	A violation of a constraint imposed by an XML values index occurred.
23526	An XML values index could not be created because the table data contains values that violate a constraint imposed by the index.
23527	Integrity constraint violated at the federated data source.
23528	A value does not conform to the data type constraint of a user-defined data type.

Table 22. Class Code 24: Invalid Cursor State

SQLSTATE	Meaning
24501	The identified cursor is not open.
24502	The cursor identified in an OPEN statement is already open.
24503	The cursor identified in the PUT statement is a select cursor, or the cursor identified in the FETCH statement is an insert cursor.
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.
24505	COMMIT is invalid, because blocking is in effect and an insert cursor is open.
24506	The statement identified in the PREPARE is the statement of an open cursor.
24507	FETCH CURRENT was specified, but the current row is deleted, or a value of an ORDER BY column of the current row has changed.
24510	An UPDATE or DELETE operation was attempted against a delete or update hole
24512	The result table does not agree with the base table.
24513	FETCH NEXT, PRIOR, CURRENT, or RELATIVE is not allowed, because the cursor position is not known.
24514	A previous error has disabled this cursor.
24516	A cursor has already been assigned to a result set.
24517	A cursor was left open by an external function or method.
24518	A cursor is not defined to handle row sets, but a rowset was requested.
24519	A hole was detected on a multiple-row FETCH statement, but indicator variables were not provided.
24520	The cursor identified in the UPDATE or DELETE statement is not positioned on a rowset.
24521	A positioned DELETE or UPDATE statement specified a row of a rowset, but the row is not contained within the current rowset.
24522	The fetch orientation is inconsistent with the definition of the cursor and whether rowsets are supported for the cursor.
24524	A FETCH CURRENT CONTINUE was requested, but there is no truncated LOB or XML data to return.
24525	An OPEN or FETCH on a cursor attempted a recursive operation on the same cursor.

Table 23. Class Code 25: Invalid Transaction State

SQLSTATE	Meaning
25000	An insert, update, or delete operation is invalid in the context where it is specified.
25001	The statement is only allowed as the first statement in a unit of work.
25006	An update operation is not valid because the transaction is read only.

SQLSTATE	Meaning
25501	The statement is only allowed as the first statement in a unit of work.
25502	Operation cannot occur multiple times in a single transaction.
25503	The federated server topology is not valid for two-phase commit transactions.

Table 24. Class Code 26: Invalid SQL Statement Identifier

SQLSTATE	Meaning
26501	The statement identified does not exist.
26505	An extended EXECUTE, DECLARE CURSOR, or DESCRIBE has been issued against an empty section.
26507	An extended EXECUTE with an OUTPUT DESCRIPTOR has been issued against a section that is not a Single Row SELECT.
26508	The statement identified in an extended PREPARE Single Row is not a select-statement.
26510	The statement name specified in a DECLARE CURSOR already has a cursor allocated to it.

Table 25. Class Code 27: Triggered Data Change Violation

SQLSTATE	Meaning
27000	An attempt was made to change the same row in the same table in the same SQL statement more than once.

Table 26. Class Code 28: Invalid Authorization Specification

SQLSTATE	Meaning
28000	Authorization name is invalid.

Table 27. Class Code 2D: Invalid Transaction Termination

SQLSTATE	Meaning
2D521	SQL COMMIT or ROLLBACK are invalid in the current operating environment.
2D522	COMMIT and ROLLBACK are not allowed in an ATOMIC Compound statement.
2D528	Dynamic COMMIT or COMMIT ON RETURN procedure is invalid for the application execution environment
2D529	Dynamic ROLLBACK is invalid for the application execution environment.

Table 28. Class Code 2E: Invalid Connection Name

SQLSTATE	Meaning
2E000	Connection name is invalid.

Table 29. Class Code 2F: SQL Function Exception

SQLSTATE	Meaning
2F002	The SQL function attempted to modify data, but the function was not defined as MODIFIES SQL DATA.

SQLSTATE	Meaning
2F003	The statement is not allowed in a function or procedure.
2F004	The SQL function attempted to read data, but the function was not defined as READS SQL DATA.
2F005	The function did not execute a RETURN statement.

Table 30. Class Code 33: Invalid SQL Descriptor Name

SQLSTATE	Meaning
33000	SQL descriptor name is invalid.

Table 31. Class Code 34: Invalid Cursor Name

SQLSTATE	Meaning
34000	Cursor name is invalid.

Table 32. Class Code 35: Invalid Condition Number

SQLSTATE	Meaning
35000	Condition number is invalid.

Table 33. Class Code 36: Cursor Sensitivity Exception

SQLSTATE	Meaning
36001	A SENSITIVE cursor cannot be defined for the specified select-statement.

Table 34. Class Code 38: External Function Exception

SQLSTATE	Meaning
38xxx	Valid error SQLSTATEs returned by an external routine or trigger.
38000	A Java routine has exited with an exception.
38001	The external routine is not allowed to execute SQL statements.
38002	The external routine attempted to modify data, but the routine was not defined as MODIFIES SQL DATA.
38003	The statement is not allowed in a routine.
38004	The external routine attempted to read data, but the routine was not defined as READS SQL DATA.
38501	Error occurred while calling a user-defined function, procedure, or trigger (using the SIMPLE CALL or SIMPLE CALL WITH NULLS calling convention).
38502	The external function is not allowed to execute SQL statements.
38503	A user-defined function or procedure has abnormally terminated (abend).
38504	A routine has been interrupted by the user.
38505	An SQL statement is not allowed in a routine on a FINAL CALL.
38506	Function failed with error from OLE DB provider.
38552	A function in the SYSFUN schema has terminated with an error.
38553	A function in a system schema has terminated with an error.

SQLSTATE	Meaning
38554	The procedure has encountered an unsupported version number for a parameter.
38H01	An MQSeries function failed to initialize.
38H02	MQSeries Application Messaging Interface failed to terminate the session.
38H03	MQSeries Application Messaging Interface failed to properly process a message.
38H04	MQSeries Application Messaging Interface failed in sending a message.
38H05	MQSeries Application Messaging Interface failed to read/receive a message.
38H06	An MQSeries Application Messaging Interface message was truncated.
38H07	MQSeries Application Messaging Interface failed to commit the unit of work.
38H08	MQSeries Application Messaging Interface policy error.
38H09	MQSeries XA (two phase commit) API call error.
38H0A	MQSeries Application Messaging Interface failed to roll back the unit of work.
38H10	Error occurred during text search processing.
38H11	Text search support is not available.
38H12	Text search is not allowed on a column because a text search index does not exist on the column.
38H13	A conflicting search administration procedure or command is running on the same text search index.
38H14	Text search administration procedure error.

Table 35. Class Code 39: External Function Call Exception

SQLSTATE	Meaning
39004	A null value is not allowed for an IN or INOUT argument when using PARAMETER STYLE GENERAL or an argument that is a Java primitive type.
39501	An output argument value returned from a function or a procedure was too long.
39502	An output SQLDA from a procedure was incorrectly modified.

Table 36. Class Code 3B: Savepoint Exception

SQLSTATE	Meaning
3B001	The savepoint is not valid.
3B002	The maximum number of savepoints has been reached.
3B501	A duplicate savepoint name was detected.
3B502	A RELEASE or ROLLBACK TO SAVEPOINT was specified, but a savepoint does not exist.
3B503	A SAVEPOINT, RELEASE SAVEPOINT, or ROLLBACK TO SAVEPOINT is not allowed in a trigger, function, or global transaction.
3B504	A SAVEPOINT is not allowed because a resource is registered that does not support savepoints.

Table 37. Class Code 3C: Ambiguous Cursor Name

SQLSTATE	Meaning
3C000	The cursor name is ambiguous.

Table 38. Class Code 3F: Invalid Schema Name

SQLSTATE	Meaning
3F000	The schema name is invalid.

Table 39. Class Code 40: Transaction Rollback

SQLSTATE	Meaning
40001	Deadlock or timeout with automatic rollback occurred.
40003	The statement completion is unknown.
40503	A private dbspace is in use by another application process.
40504	A system error has caused the unit of work to be rolled back.
40506	The current transaction was rolled back because of an SQL error.
40507	The current transaction was rolled backed as a result of a failure creating an index.
40508	The current transaction was rolled backed as a result of an error updating a continuously available table. The current transaction was rolled backed as a result of an error updating a continuously available table.
4050A	A data source connection was terminated during the processing of a federated transaction.

Table 40. Class Code 42: Syntax Error or Access Rule Violation

SQLSTATE	Meaning
42501	The authorization ID does not have the privilege to perform the specified operation on the identified object.
42502	The authorization ID does not have the privilege to perform the operation as specified.
42503	The specified authorization ID or one of the authorization IDs of the application process is not allowed.
42504	A specified privilege, security label, or exemption cannot be revoked from a specified authorization-name.
42505	Connection authorization failure occurred.
42506	Owner authorization failure occurred.
42508	The specified database privileges cannot be granted to PUBLIC.
42509	SQL statement is not authorized, because of the DYNAMICRULES option.
42510	The authorization ID does not have the privilege to create functions or procedures in the WLM environment.
42511	The authorization ID does not have the privilege to retrieve the DATALINK value.
42512	The authorization ID does not have security to the protected column.
42513	The authorization ID does not have the MLS WRITE-DOWN privilege.
42514	The authorization ID does not have the privileges necessary for ownership of the object.
42516	Authentication at the user mapping repository failed.
42517	The specified authorization ID is not allowed to use the trusted context.
42519	This authorization ID is not allowed to perform the operation on the protected table.
42520	A built-in function could not be executed because the authorization ID does not have a security label.
42521	The authority or privilege cannot be granted to the specified authorization ID.
42522	The authorization ID does not have the credentials to protect a column or remove protection from a column.

SQLSTATE	Meaning
42523	An attempt to revoke SECADM authority was denied because the authorization ID is the only external authorization ID with SECADM.
42524	The current session user does not have usage privilege on any enabled workloads.
42525	The statement failed because of an authorization error from a Big SQL component.
42601	A character, token, or clause is invalid or missing.
42602	A character that is invalid in a name has been detected.
42603	An unterminated string constant has been detected.
42604	An invalid numeric or string constant has been detected.
42605	The number of arguments specified for a scalar function is invalid.
42606	An invalid hexadecimal constant has been detected.
42607	An operand of an aggregate function or CONCAT operator is invalid.
42608	The use of NULL or DEFAULT in VALUES or an assignment statement is invalid.
42609	All operands of an operator or predicate are parameter markers.
42610	A parameter marker or the null value is not allowed.
42611	The column, argument, parameter, or global variable definition is invalid.
42612	The statement string is an SQL statement that is not acceptable in the context in which it is presented.
42613	Clauses are mutually exclusive.
42614	A duplicate keyword or clause is invalid.
42615	An invalid alternative was detected.
42616	Invalid options are specified.
42617	The statement string is blank or empty.
42618	A variable is not allowed.
42620	Read-only SCROLL was specified with the UPDATE clause.
42621	The check constraint or generated column expression is invalid.
42622	A name or label is too long.
42623	A DEFAULT clause cannot be specified.
42625	A CASE expression is invalid.
42626	A column specification is not allowed for a CREATE INDEX that is built on an auxiliary table.
42627	RETURNS clause must be specified prior to predicate specification using the EXPRESSION AS clause.
42628	A TO SQL or FROM SQL transform function is defined more than once in a transform definition.
42629	Parameter names must be specified for SQL routines.
42630	An SQLSTATE or SQLCODE variable is not valid in this context.
42631	An expression must be specified on a RETURN statement in an SQL function.
42632	There must be a RETURN statement in an SQL function or method.
42633	An AS clause is required for an argument of XMLATTRIBUTES or XMLFOREST.
42634	The XML name is not valid.
42635	The XML namespace prefix is not valid.
42636	The BY REF or BY VALUE clause is missing or used incorrectly.
42637	An XQuery expression cannot be specified in a DECLARE CURSOR statement.
42638	An obfuscated statement is not valid.

SQLSTATE	Meaning
42701	The same target is specified more than once for assignment in the same SQL statement.
42702	A column reference is ambiguous, because of duplicate names.
42703	An undefined column or parameter name was detected.
42704	An undefined object or constraint name was detected.
42705	An undefined server-name was detected.
42706	Column names in ORDER BY are invalid, because all columns of the result table are unnamed.
42707	A column name in ORDER BY does not identify a column of the result table.
42708	The locale specified in a SET LOCALE or locale sensitive function was not found.
42709	A duplicate column name was specified in a key column list.
42710	A duplicate object or constraint name was detected.
42711	A duplicate column name was detected in the object definition or ALTER TABLE statement.
42712	A duplicate table designator was detected in the FROM clause or REFERENCING clause of a CREATE TRIGGER statement.
42713	A duplicate object was detected in a list or is the same as an existing object.
42714	A host variable can be defined only once.
42716	Any function called within the body of an inline function must already be defined.
42718	The local server name is not defined.
42720	The nodename for the remote database was not found in the node directory.
42721	The special register name is unknown at the server.
42723	A routine with the same signature already exists in the schema, module, or compound block where it is defined.
42724	Unable to access an external program used for a user-defined function or a procedure.
42725	A routine was referenced directly (not by either signature or by specific instance name), but there is more than one specific instance of that routine.
42726	Duplicate names for common table expressions were detected.
42727	No default primary table space exists for the new table.
42728	A duplicate member number or database partition number was detected in the list of member numbers or database partition numbers.
42729	The specified member number or database partition number is not valid.
42730	The container name is already used by another table space.
42731	The container name is already used by this table space.
42732	A duplicate schema name in a special register was detected.
42733	A procedure with the specified name cannot be added to the schema because the procedure overloading is not allowed in this database and there is already a procedure with the same name in the schema.
42734	A duplicate parameter-name, SQL variable name, label, or condition-name was detected.
42735	The database partition group for the table space is not defined for the buffer pool.
42736	The label specified on the GOTO, ITERATE, or LEAVE statement is not found or not valid.
42737	The condition specified is not defined.
42738	A duplicate column name or unnamed column was specified in a DECLARE CURSOR statement of a FOR statement.
42739	A duplicate transform was detected.
42740	No transforms were found for the specified type. No transforms were dropped.

SQLSTATE	Meaning
42741	A transform group is not defined for a data type.
42742	A subtable or subview of the same type already exists in the typed table or typed view hierarchy.
42743	The search method is not found in the index extension.
42744	A TO SQL or FROM SQL transform function is not defined in a transform group.
42745	The routine would define a overriding relationship with an existing method.
42746	A method name cannot be the same as a structured type name within the same type hierarchy.
42747	The same descriptor item was specified more than once in the same SET DESCRIPTOR statement.
42748	The storage path already exists for the database or is specified more than once.
42749	An XML schema document with the same target namespace and schema location already exists for the XML schema.
4274A	An XSROBJECT is not found in the XML schema repository.
4274B	A unique XSROBJECT could not be found in the XML schema repository.
4274C	The specified attribute was not found in the trusted context.
4274D	The specified attribute already exists in the trusted context.
4274E	The specified attribute is not supported in the trusted context.
4274F	The component element is not defined in the security label component.
4274G	The security label component is not defined in the security policy used by the given security label.
4274H	The specified access rule does not exist for the specified security policy.
4274I	The security label does not exist for the specified security policy.
4274J	The database partition group is already used by this buffer pool.
4274K	Invalid use of a named argument when invoking a routine.
4274L	The database partition group is already used by this buffer pool.
4274M	An undefined period name was detected.
42801	Isolation level UR is invalid, because the result table is not read-only.
42802	The number of insert or update values is not the same as the number of columns or variables.
42803	A column reference in the SELECT or HAVING clause is invalid, because it is not a grouping column; or a column reference in the GROUP BY clause is invalid.
42804	The result expressions in a CASE expression are not compatible.
42805	An integer in the ORDER BY clause does not identify a column of the result table.
42806	A value cannot be assigned to a variable, because the data types are not compatible.
42807	The data-change statement is not permitted on this object.
42808	A column identified in the INSERT or UPDATE operation is not updatable.
42809	The identified object is not the type of object to which the statement applies.
42810	A base table is not identified in a FOREIGN KEY clause.
42811	The number of columns specified is not the same as the number of columns in the SELECT clause.
42812	A library name is required in CREATE TABLE in the system naming mode.
42813	WITH CHECK OPTION cannot be used for the specified view.
42814	The column cannot be dropped, because it is the only column in the table.
42815	The data type, length, scale, value, or CCSID is invalid.
42816	A datetime value or duration in an expression is invalid.
42817	The column cannot be dropped because a view or constraint is dependent on the column, the column is part of a partitioning key, or is a security label column.

SQLSTATE	Meaning
42818	The operands of an operator or function are not compatible or comparable.
42819	An operand of an arithmetic operation or an operand of a function that requires a number is invalid.
42820	A numeric constant is too long, or it has a value that is not within the range of its data type.
42821	A data type for an assignment to a column or variable is not compatible with the data type.
42822	An expression in the ORDER BY clause or GROUP BY clause is not valid.
42823	Multiple columns are returned from a subquery that only allows one column.
42824	An operand of LIKE is not a string, or the first operand is not a column.
42825	The rows of UNION, INTERSECT, EXCEPT, or VALUES do not have compatible columns.
42826	The rows of UNION, INTERSECT, EXCEPT, or VALUES do not have the same number of columns.
42827	The table identified in the UPDATE or DELETE is not the same table designated by the cursor.
42828	The table designated by the cursor of the UPDATE or DELETE statement cannot be modified, or the cursor is read-only.
42829	FOR UPDATE OF is invalid, because the result table designated by the cursor cannot be modified.
42830	The foreign key does not conform to the description of the parent key.
42831	Null values are not allowed in a column of a primary key, a column of a unique key, a ROWID column, a row change timestamp column, a row-begin column, a row-end column, or a column of an application period.
42832	The operation is not allowed on system objects.
42833	The qualified object name is inconsistent with the naming option.
42834	SET NULL cannot be specified, because no column of the foreign key can be assigned the null value.
42835	Cyclic references cannot be specified between named derived tables.
42836	The specification of a recursive, named derived table is invalid.
42837	The column cannot be altered, because its attributes are not compatible with the current column attributes.
42838	An invalid use of a table space was detected.
42839	Indexes and long columns cannot be in separate table spaces from the table.
42841	An untyped expression cannot be a user-defined type or reference type.
42842	A column or parameter definition is invalid, because a specified option is inconsistent with the column description.
42844	A function in a select list item has produced a BOOLEAN result.
42845	An invalid use of a NOT DETERMINISTIC or EXTERNAL ACTION function was detected.
42846	Cast from source type to target type is not supported.
42847	An OVRDBF command was issued for one of the referenced files, but one of the parameters is not valid for SQL.
42848	Isolation level CS WITH KEEP LOCKS is not allowed.
42849	The specified option is not supported for routines.
42850	A logical file is invalid in CREATE VIEW.
42851	A referenced file is not a table, view, or physical file.
42852	The privileges specified in GRANT or REVOKE are invalid or inconsistent. (For example, GRANT ALTER on a view.)
42853	Both alternatives of an option were specified, or the same option was specified more than once.

SQLSTATE	Meaning
42854	A result column data type in the select list is not compatible with the defined type in a typed view or materialized query table definition.
42855	The assignment of the LOB or XML to this variable is not allowed. The target variable for all fetches of a LOB or XML value for this cursor must be the same for all FETCHes.
42856	The alter of a CCSID to the specified CCSID is not valid.
42857	A referenced file has more than one format or data space.
42858	Operation cannot be applied to the specified object.
42860	The constraint cannot be dropped because it is enforcing a primary key or ROWID.
42862	An extended dynamic statement cannot be executed against a non-extended dynamic package.
42863	An undefined host variable in REXX has been detected.
42866	The data type in either the RETURNS clause or the CAST FROM clause in the CREATE FUNCTION statement is not appropriate for the data type returned from the sourced function or RETURN statement in the function body.
42867	Conflicting options have been specified.
42872	FETCH statement clauses are incompatible with the cursor definition.
42873	An invalid number of rows was specified in a multiple-row FETCH or multiple-row INSERT.
42874	ALWCPYDTA(*NO) was specified, but a copy is necessary to implement the select-statement.
42875	The schema-name portion of a qualified name must be the same name as the schema name.
42876	Different CCSIDs for keys in CREATE INDEX are only allowed with a *HEX collating sequence.
42877	The column name cannot be qualified.
42878	An invalid function or procedure name was used with the EXTERNAL keyword.
42879	The data type of one or more input parameters in the CREATE FUNCTION statement is not appropriate for the corresponding data type in the source function.
42880	The CAST TO and CAST FROM data types are incompatible, or would always result in truncation of a fixed string.
42881	Invalid use of a function.
42882	The specific instance name qualifier is not equal to the function name qualifier.
42883	No routine was found with a matching signature.
42884	No routine was found with the specified name and compatible arguments.
42885	The number of input parameters specified on a CREATE FUNCTION statement does not match the number provided by the function named in the SOURCE clause.
42886	The IN, OUT, or INOUT parameter attributes do not match.
42887	The function is not valid in the context where it occurs.
42888	The table does not have a primary key.
42889	The table already has a primary key.
42890	A column list was specified in the references clause, but the identified parent table does not have a unique constraint with the specified column names.
42891	A duplicate UNIQUE constraint already exists.
42892	The referential constraint and trigger are not allowed, because the DELETE rule and trigger event are not compatible.
42893	The object or constraint cannot be dropped, altered, or transferred or authorities cannot be revoked from the object, because other objects are dependent on it.
42894	The value of a column or sequence attribute is invalid.

SQLSTATE	Meaning
42895	For static SQL, an input variable cannot be used, because its data type is not compatible with the parameter of a procedure or user-defined function.
42896	The ASP number is invalid.
42898	An invalid correlated reference or transition table was detected in a trigger.
42899	Correlated references and column names are not allowed for triggered actions with the FOR EACH STATEMENT clause.
428A0	An error occurred with the sourced function on which the user-defined function is based.
428A1	Unable to access a file referenced by a file reference variable.
428A2	The table created in the multi-partition database partition group because no column exists that can be used as the distribution key.
428A3	An invalid path has been specified for an event monitor.
428A4	An invalid value has been specified for an event monitor option.
428A5	An exception table named in a SET INTEGRITY statement either does not have the proper structure, or it has been defined with generated columns, constraints or triggers.
428A6	An exception table named in a SET CONSTRAINTS statement cannot be the same as one of the tables being checked.
428A7	There is a mismatch in the number of tables being checked and in the number of exception tables specified in the SET CONSTRAINTS statement.
428A8	Cannot reset the set-integrity-pending state using the SET INTEGRITY statement on a descendent table while a parent table or underlying table is in the set-integrity-pending state.
428A9	The specified member number or database partition number, or range of member numbers or database partition numbers is invalid.
428AA	The column name is not a valid column for an event monitor table.
428B0	Nesting not valid in ROLLUP, CUBE, or GROUPING SETs.
428B1	The clause specifying table space containers that are not designated for specific database partitions is either missing or specified more than once.
428B2	The path name for the container is not valid.
428B3	An invalid SQLSTATE was specified.
428B4	The part clause of a LOCK TABLE statement is not valid.
428B7	A number specified in an SQL statement is out of the valid range.
428B8	The name specified on a rename is not valid.
428BA	WITHOUT RETURN cursors must not be specified in SET RESULT SETS.
428C0	The database partition cannot be dropped, because it is the only database partition in the database partition group.
428C1	The data type or attribute of a column can only be specified once for a table.
428C2	Examination of the function body indicates that the given clause should have been specified on the CREATE FUNCTION statement.
428C3	The language specified for a subtype must be the same as that of its supertype.
428C4	The number of elements on each side of the predicate operator is not the same.
428C5	No data type mapping was found for a data type from the data source.
428C7	A ROWID or reference column specification is not valid or used in an invalid context.
428C9	A column defined as GENERATED ALWAYS cannot be specified as the target column of an insert or update operation.
428CA	A table in append mode cannot have a clustered index.

SQLSTATE	Meaning
428CB	The pagesize for a table space must match the page size of the associated buffer pool.
428D1	Unable to access a file referenced by a DATALINK value.
428D2	AS LOCATOR cannot be specified for a non-LOB parameter.
428D3	GENERATED is not allowed for the specified data type or attribute of a column.
428D4	A cursor specified in a FOR statement cannot be referenced in an OPEN, CLOSE, or FETCH statement.
428D5	The ending label does not match the beginning label.
428D6	UNDO is not allowed for NOT ATOMIC compound statements.
428D7	The condition value is not allowed.
428D8	The sqlcode or sqlstate variable declaration is not valid.
428D9	The table specified in the variable in the LIKE clause is not compatible with the table specified in the LIKE clause.
428DB	An object is not valid as a supertype, supertable, or superview.
428DC	The function or method is not valid as the transform for this type.
428DE	The PAGESIZE value is not supported.
428DF	The data types specified in CREATE CAST are not valid.
428DG	The function specified in CREATE CAST is not valid.
428DH	The operation is not valid for typed tables.
428DJ	The options associated with an inherited column cannot be changed.
428DK	The scope for the reference column is already defined.
428DL	The parameter of an external or sourced function has a scope defined.
428DM	The scope table or view is not valid for the reference type.
428DN	SCOPE is not specified in the RETURNS clause of an external function or is specified in the RETURNS clause of a sourced function.
428DP	The type is not a structured type.
428DQ	A subtable or subview cannot have a different schema name than its supertable or superview.
428DR	Operation cannot be applied to a subtable or a subview.
428DS	An index on the specified columns cannot be defined on the subtable.
428DT	The operand of an expression is not a valid scoped reference type.
428DU	A type is not included in the required type hierarchy.
428DV	The left operand of a dereference operator is not valid.
428DW	Object identifier column cannot be referenced using the dereference operator.
428DX	Object identifier column is required to define the root table or root view of a typed table or typed view hierarchy.
428DY	Statistics cannot be updated for the target object type.
428DZ	An object identifier column cannot be updated.
428E0	The definition of the index does not match the definition of the index extension.
428E1	The result of the range-producing table function is inconsistent with that of the key transformation table function for the index extension.
428E2	The number or the type of key-target parameters does not match the number or type of key transform function for the index extension.
428E3	The argument for the function in an index extension is not valid.

SQLSTATE	Meaning
428E4	The function is not supported in an CREATE INDEX EXTENSION statement.
428E5	SELECTIVITY clause can only be specified with a user-defined predicate.
428E6	The argument of the search method in the user-defined predicate does not match the one in the corresponding search method of the index extension.
428E7	The type of the operand following the comparison operator in the user-defined predicate does not match the RETURNS data type.
428E8	A search target or search argument parameter does not match a parameter name of the function being created.
428E9	An argument parameter name cannot appear as both a search target and search argument in the same exploitation rule.
428EA	A fullselect in a typed view is not valid.
428EB	A column in a subview cannot be read only when the corresponding column in the superview is updatable.
428EC	The fullselect specified for the materialized query table is not valid.
428ED	Structured types with Datalink or Reference type attributes cannot be constructed.
428EE	Option not valid for remote data source.
428EF	Value for the option is not valid for the this data source.
428EG	Missing required option for remote data source.
428EH	Option is already defined for remote data source.
428EJ	Option is not defined so cannot be set for remote data source.
428EK	The schema qualifier is not valid.
428EL	A transform function not valid for use with a function or method.
428EM	The TRANSFORM GROUP clause is required.
428EN	A transform group is specified that is not used.
428EP	A structured type cannot depend on itself either directly or indirectly.
428EQ	The returns type of the routine is not the same as the subject type.
428ER	A method specification cannot be dropped before the method body is dropped.
428ES	A method body does not correspond to the language type of the method specification.
428ET	INLINE LENGTH value is not valid.
428EU	TYPE or VERSION is not specified in the server definition.
428EV	Pass-through facility is not supported for the type of data source.
428EW	The table cannot be converted to or from a materialized query table.
428EX	Routine cannot be used as a transform function because it is either a built-in function or a method.
428EY	The data type of the search target in a user-defined predicate does not match the data type of the source key of the specified index extension.
428EZ	A window specification for an OLAP function is not valid.
428F0	A ROW function must include at least two columns.
428F1	An SQL TABLE function must return a table result.
428F2	An integer expression must be specified on a RETURN statement in an SQL procedure.
428F4	The SENSITIVITY specified on FETCH is not allowed for the cursor.
428F5	The invocation of a routine is ambiguous.
428F7	The operation was attempted on an external routine, but the operation is only allowed on an SQL routine.

SQLSTATE	Meaning
428F9	A sequence expression cannot be specified in this context.
428FA	The scale of the decimal number must be zero.
428FB	Sequence-name must not be a sequence generated by the system.
428FC	The length of the encryption password is not valid.
428FD	The password used for decryption does not match the password used to encrypt the data.
428FE	The data is not a result of the ENCRYPT function.
428FF	The buffer pool specification is not valid.
428FG	The staging table or materialized query table definition is not valid.
428FH	The materialized query table option is not valid.
428FI	The ORDER OF clause was specified, but the referenced table designator is not ordered.
428FJ	ORDER BY or FETCH FIRST is not allowed in the outer fullselect of a view or materialized query table.
428FL	A data change statement is not allowed in the context in which it was specified.
428FM	An SQL data change statement within a SELECT specified a view which is not a symmetric view.
428FN	ALLOW FULL REFRESH must be specified for altering this view.
428FO	The ALTER VIEW failed because the fullselect is invalid.
428FP	Only one INSTEAD OF trigger is allowed for each kind of operation on a view.
428FQ	An INSTEAD OF trigger cannot be created because of how the view is defined.
428FR	A column cannot be altered as specified.
428FS	A column cannot be added to an index.
428FT	The partitioning clause specified on CREATE or ALTER is not valid.
428FU	Built-in type returned from the FROM SQL transform function or method does not match the corresponding built-in type for the TO SQL transform function or method.
428FV	The method cannot be defined as an overriding method.
428FW	The server options are either invalid or missing for continuous availability.
428FX	The column definitions for the local table do not match the column definitions of the specified remote object.
428FY	A column cannot be added, dropped, or altered in a materialized query table.
428FZ	The view is the target in the MERGE statement, but is missing the INSTEAD OF trigger for the operation.
428G0	A logical file prevents the alter.
428G1	The number of data partitions exceeds the number of table spaces for the table.
428G2	The last data partition cannot be dropped from the table.
428G3	FINAL TABLE is not valid when the target view of the SQL data change statement in a fullselect has an INSTEAD OF trigger defined.
428G4	Invalid use of INPUT SEQUENCE ordering.
428G5	The assignment clause of the UPDATE statement must specify at least one column that is not an INCLUDE column.
428G6	A column is specified that cannot be selected from the target of the data change statement in the FROM clause of the fullselect.
428G7	A nickname cannot be referenced in an enforced referential constraint.
428G8	The view cannot be enabled for query optimization.
428G9	Wrapper template substitution error.

SQLSTATE	Meaning
428GA	Federated option cannot be added, dropped, or altered.
428GB	A character could not be converted and substitution characters are not allowed.
428GC	An invalid string unit was specified for a function.
428GE	The source table cannot be attached to the partitioned target table.
428GF	The GRANT roles statement is not valid as it would create a cycle.
428GG	Invalid use of an error tolerant nested-table-expression.
428GH	The data type of one or more parameters specified in the ADD VERSION clause does not match the corresponding data type in the routine being altered.
428GI	An XML schema is not complete because an XML schema document is missing.
428GJ	The table cannot be truncated because DELETE triggers exist for the table or the table is a parent table of a referential constraint that would be affected by the statement.
428GK	An ALTER TRUSTED CONTEXT attempted to remove one or more of the minimum required attributes.
428GL	The system authorization ID specified for a trusted context is already specified in another trusted context.
428GM	The trusted context is already defined to be used by this authorization ID or PUBLIC.
428GN	The specified authorization ID or PUBLIC is not defined in the specified trusted context.
428GO	A column option is invalid in a transparent DDL statement.
428GP	Multiple elements cannot be specified for a security label component of type ARRAY.
428GQ	The GRANT of the security label conflicts with an existing granted security label for the component.
428GR	A security label with the same access type (READ or WRITE) has already been granted to the authorization ID.
428GS	The option value specified for the procedure does not match the corresponding option of the source procedure.
428GT	The table is not protected with a security policy.
428GU	A table must include at least one column that is not implicitly hidden.
428GV	URI is an empty string.
428GW	The operation is not allowed on strings that are not FOR BIT DATA.
428GX	A global variable cannot be set or referenced in this context.
428GZ	All specified instances of sort-keys in the SELECT clause are not identical.
428H0	The array index cannot be applied to an object whose type is not ARRAY.
428H1	The data type of an array index expression is not valid for the array.
428H2	Data type is not supported in the context where it is being used.
428H3	The tree element is not valid where specified.
428H4	A hierarchical query construct is used out of context.
428H5	The specified join operation is not valid.
428H6	XMLPARSE is not allowed for binary XML.
428H7	Expression cannot be computed as a single value for the query.
428H8	The object must be defined as secure because another object depends on it for row-level or column-level access control.
428H9	PERMISSION or MASK cannot be altered.

SQLSTATE	Meaning
428HA	An argument of a user-defined table function must not reference a column for which a column mask is defined.
428HB	A permission or mask cannot be created on the specified object.
428HC	A column mask is already defined for the specified column.
428HD	The statement cannot be processed because a column mask cannot be applied or the definition of the mask conflicts with the statement.
428HE	The user mapping or federated server option conflicts with an existing user mapping or federated server option.
428HF	The invocation of the routine omits a parameter which is not defined with a DEFAULT.
428HG	Parameter definition for the routine includes an option that is invalid in the context where it is used.
428HH	The drop storage path failed because the database must have at least one storage path associated with it.
428HI	The table function for the specified procedure cannot be created because the attributes of the procedure are incompatible.
428HJ	The organization clause specified on CREATE or ALTER is not valid.
428HK	The specified hash space is not valid for the implicitly created table space.
428HM	The system versioning clause specified on CREATE or ALTER is not valid.
428HN	The period specification is not valid.
428HP	The definition of a module initialization procedure SYS_INIT is not valid.
428HQ	Invalid data type for the operand of a predicate.
428HR	Invalid use of row data type value in a list of values.
428HS	The target object of the anchored data type is not supported in the context where it is used.
428HT	The WITH ORDINALITY clause is not valid with the specified UNNEST argument.
428HU	Invalid use of dynamic statement name in a cursor value constructor.
428HV	Error occurred processing a conditional compilation directive.
428HW	The period specification in an index or constraint is not valid.
428HX	The table is not valid for a history table or archive table.
428HY	The period specification or period condition is not valid.
428HZ	The temporal attribute of the table was not valid for the specified ALTER operation.
428I0	A variable is the target of two or more assignments with no defined order of assignment.
428I1	The columns updated by the XMLMODIFY function were not specified in the UPDATE SET clause.
428I2	A clause is not supported for in a transparent DDL statement.
428I3	A global variable identified as the target of an assignment is a read only global variable.
428I4	The combination of UNNEST arguments are not valid.
428I5	The attributes of an object at one location do not match the attributes of the same object at another location.
428I6	The archive enabled table is not allowed in this context.
428I7	The columns of the primary key for the replication-maintained materialized query table do not correspond to columns of a unique or primary key constraint of the base table.
42902	The object of the INSERT, UPDATE, or DELETE is also identified (possibly implicitly through a view) in a FROM clause.
42903	Invalid use of an aggregate function or OLAP function.

SQLSTATE	Meaning
42904	The SQL procedure was not created because of a compile error.
42905	DISTINCT is specified more than once in a subselect.
42906	An aggregate function in a subquery of a HAVING clause includes an expression that applies an operator to a correlated reference.
42907	The string is too long in the context it was specified.
42908	The statement does not include a required column list.
42909	CREATE VIEW includes an operator or operand that is not valid for views.
42910	The statement is not allowed in a Compound statement.
42911	A decimal divide operation is invalid, because the result would have a negative scale.
42912	A column cannot be updated, because it is not identified in the UPDATE clause of the select-statement of the cursor.
42913	An UPDATE or DELETE WHERE CURRENT OF that is invalid has been detected.
42914	The DELETE is invalid, because a table referenced in a subquery can be affected by the operation.
42915	An invalid referential constraint has been detected.
42916	The alias cannot be created, because it would result in a repetitive chain of aliases.
42917	The object cannot be explicitly dropped or altered.
42918	A user-defined data type cannot be created with a system-defined data type name (for example, INTEGER).
42921	Containers cannot be added to the table space.
42922	DROP SCHEMA cannot be executed under commitment control.
42923	Program or package must be recreated to reference an alias-name.
42924	An alias resolved to another alias rather than a table or view at the remote location.
42925	Recursive named derived tables cannot specify SELECT DISTINCT and must specify UNION ALL.
42926	Locators are not allowed with COMMIT(*NONE).
42927	The function cannot be altered to NOT DETERMINISTIC or EXTERNAL ACTION because it is referenced by one or more existing views.
42928	WITH EMPTY TABLE cannot be specified.
42929	FOR ALL PARTITIONS is not allowed for an encoded vector index.
42930	The same column was identified in FOR UPDATE OF and ORDER BY.
42932	The program preparation assumptions are incorrect.
42937	The parameter must not have a subtype of mixed.
42939	The name cannot be used, because the specified identifier is reserved for system use.
42943	An empty non-modifiable package cannot be committed.
42944	The authorization ID cannot be both an owner and primary group owner.
42945	ALTER CCSID is not allowed on a table space or database that contains a view.
42961	The server name specified does not match the current server.
42962	A long column, LOB column, structured type column or datalink column cannot be used in an index, a key, generated column, or a constraint.
42963	Invalid specification of a security label column.
42968	The connection failed, because there is no current software license.
42969	The package was not created.
42970	COMMIT HOLD or ROLLBACK HOLD is only allowed to a DB2 for i application server.

SQLSTATE	Meaning
42971	SQL statements cannot be executed under commitment control, because commitment control is already active to another relational database.
42972	An expression in a join-condition or ON clause of a MERGE statement references columns in more than one of the operand tables.
42977	The authorization ID cannot be changed when connecting to the local server.
42978	An indicator variable is not a small integer.
42981	CREATE SCHEMA is not allowed if changes are pending in the unit of work.
42984	The privilege cannot be granted to the view, because *OBJOPR or *OBJMGT authority exists on a dependent view or table, and the grantee does not have *ALLOBJ or the specified privilege on the dependent table or view.
42985	The statement is not allowed in a routine.
42986	The source table of a rename operation is referenced in a context where is it not supported.
42987	The statement or routine is not allowed in a trigger.
42988	The operation is not allowed with mixed ASCII data.
42989	A column that is generated using on an expression, or as a row change timestamp, or is a security label column cannot be used in a BEFORE trigger.
42990	A unique index or unique constraint is not allowed because the key columns are not a superset of the partitioned key columns.
42991	The BOOLEAN data type is currently only supported internally.
42993	The column, as defined, is too large to be logged.
42994	Raw device containers are not supported.
42995	The requested function does not apply to global temporary tables.
42996	The partition key cannot be a datetime or floating-point column.
42997	Capability is not supported by this version of the DB2 application requester, DB2 application server, or the combination of the two.
42998	A referential constraint is not allowed because the foreign key columns are not a superset of the partitioned key columns or the node group is not the same as the parent table.
42999	Function not supported for query.
429A0	A foreign key cannot reference a parent table defined as "not logged initially".
429A1	The database partition group is not valid for the table space.
429A2	The row type within a row reference must be the same as the row type of the target table.
429A3	Row type cannot be directly used as the type of a column. Only references to row types are allowed.
429A5	A row type can not be added to a table that already has a row type.
429A6	The table identified in a row reference operation does not have a row type.
429A7	Function with the READS SQL DATA property cannot be used in the specified context.
429A8	Distinct types cannot be based on either REF types or ADTs.
429A9	SQL statement cannot be processed by DataJoiner.
429AA	The "not logged initially" attribute cannot be activated.
429B1	A procedure specifying COMMIT ON RETURN cannot be the target of a nested CALL statement.
429B2	The specified inline length value for the structured type or column is too small.
429B3	The object may not be defined on a subtable.
429B4	The data filter function cannot be a LANGUAGE SQL function.

SQLSTATE	Meaning
429B5	The data type of the instance parameter in the index extension is not valid in the same exploitation rule.
429B6	Rows from a distributed table cannot be redistributed because the table contains a datalink column with FILE LINK CONTROL.
429B7	A referential constraint with a delete rule of CASCADE is not allowed on a table with a DataLink column with FILE LINK CONTROL.
429B8	A routine defined with PARAMETER STYLE JAVA cannot have a structured type as a parameter or returns type.
429B9	DEFAULT or NULL cannot be used in an attribute assignment.
429BA	The FEDERATED keyword must be used with a reference to a federated database object.
429BB	The data type of a column, parameter, or SQL variable is not supported.
429BC	There are multiple conflicting container operations in the ALTER TABLESPACE statement.
429BD	RETURN must be the last SQL statement of the atomic compound statement within an SQL row or table function.
429BE	The primary key or a unique key is a subset of the columns in the dimensions clause.
429BF	The table cannot be truncated.
429BG	The function is not supported on range-clustered tables.
429BH	A partitioned table cannot contain an unsupported column definition such as a datalink column or XML column.
429BI	The condition area is full and cannot handle more errors for a NOT ATOMIC statement.
429BJ	Invalid usage of WITH ROW MOVEMENT in a view.
429BK	Invalid attempt to update a view because of row movement involving underlying views.
429BL	A function which modifies SQL data is invoked in an illegal context.
429BM	The collating sequence cannot be used in this context.
429BN	A CREATE statement cannot be processed when the value of CURRENT SCHEMA differs from CURRENT SQLID.
429BO	No plan was possible to create for the federated data source.
429BP	Invalid nickname column expression.
429BQ	The specified alter of the data type or attribute is not allowed.
429BS	Invalid index definition involving an XMLPATTERN clause or a column defined with a data type of XML.
429BT	Transfer ownership failed because of a dependency.
429BU	The user mappings from the user mapping repository for a plugin cannot be accessed.
429BV	Invalid specification of a ROW CHANGE TIMESTAMP column.
429BW	The statement cannot be processed due to related implicitly created objects.
429BX	The expression for an index key is not valid.
429BY	The statement is not allowed when using a trusted connection.
429BZ	Insert into a UNION ALL view failed because one of the underlying tables is protected.
429C0	The query must contain a predicate using the indicated column.
429C1	A data type cannot be determined for an untyped parameter marker.
429C2	The data type specified for an array is not valid in the context where it is specified.
429C3	The creation or revalidation of an object would result in an invalid direct or indirect self-reference.
429C4	Revalidation failed for all objects that were specified to be revalidated.

SQLSTATE	Meaning
429C5	Data type is not supported for a field in a row type.
429C6	Invalid context of use for a compound SQL (compiled) statement.
429C8	Data definition operations are not allowed on procedures with the same name as the connect procedure.
429C9	The operation cannot be processed because it involves a text index.
429CA	ANALYZE_TABLE expression is not supported in the context where it is specified.
429CB	The attributes of the table or column are not supported for the table type.

Table 41. Class Code 44: WITH CHECK OPTION Violation

SQLSTATE	Meaning
44000	The INSERT or UPDATE is not allowed, because a resulting row does not satisfy the view definition.

Table 42. Class Code 46: Java Errors

SQLSTATE	Meaning
46001	The URL specified on an install or replace of a jar procedure did not identify a valid jar file.
46002	The jar name specified on the install, replace, or remove of a Java procedure is not valid.
46003	The jar file cannot be removed, a class is in use by a procedure.
46007	A Java function has a Java method with an invalid signature.
46008	A Java function could not map to a single Java method.
4600C	The jar cannot be removed. It is in use.
4600D	The value provided for the new Java path is invalid.
4600E	The alter of the jar failed because the specified path references itself.
46103	A Java routine encountered a ClassNotFound exception.
46501	The install or remove jar procedure specified the use of a deployment descriptor.
46502	A user-defined procedure has returned a DYNAMIC RESULT SET of an invalid class. The parameter is not a DB2 result set.

Table 43. Class Code 51: Invalid Application State

SQLSTATE	Meaning
51002	The package corresponding to an SQL statement execution request was not found.
51003	Consistency tokens do not match.
51004	An address in the SQLDA is invalid.
51005	The previous system error has disabled this function.
51006	A valid connection has not been established.
51008	The release number of the program or package is not valid.
51009	COMMIT or ROLLBACK is not allowed, because commitment control has not been started.
51010	The programmable interface for operator commands is not valid when within a unit of work.
51012	The index has been marked invalid.
51013	An attempt has been made to use an index that has been marked invalid.

SQLSTATE	Meaning
51015	An attempt was made to execute a section that was found to be in error at bind time.
51016	A package or view cannot be rebound, because the character set under which it was originally prepared is different than the character set under which the database manager is running.
51017	The user is not logged on.
51021	SQL statements cannot be executed until the application process executes a rollback operation.
51022	A CONNECT that specifies an authorization name is invalid when a connection (either current or dormant) already exists to the server named in that CONNECT statement.
51023	The database is already in use by another instance of the database manager.
51024	An object cannot be used, because it has been marked inoperative.
51025	An application in the XA transaction processing environment is not bound with SYNCPOINT TWOPHASE.
51026	An event monitor cannot be turned on, because its target path is already in use by another event monitor.
51027	The IMMEDIATE CHECKED option of the SET INTEGRITY statement is not valid since a table is not in the set-integrity-pending state.
51028	A package cannot be used, because it is marked inoperative.
51030	The procedure referenced in a DESCRIBE PROCEDURE or ASSOCIATE LOCATOR statement has not yet been called within the application process.
51032	A valid CCSID has not yet been specified for this DB2 for z/OS subsystem.
51033	The operation is not allowed because it operates on a result set that was not created by the current server.
51034	The routine defined with MODIFIES SQL DATA is not valid in the context in which it is invoked.
51035	A PREVIOUS VALUE expression cannot be used because a value has not been generated for the sequence yet in this session.
51036	An implicit connect to a remote server is not allowed because a savepoint is outstanding.
51037	The operation is not allowed because a trigger has been marked inoperative.
51038	SQL Statements may no longer be issued by the routine.
51039	The ENCRYPTION PASSWORD value is not set.
51040	Invalid compilation environment.
51041	The SQL statement cannot be issued within an XA transaction.
51042	Statistics could not be collected because there is no active statistics event monitor.
51044	The cursor variable could not be used in an OPEN statement in the current scope because the cursor constructor value was assigned in a different scope.
51045	The request is not supported for a read-only database.
51046	The data change operation is not allowed for the target object when CURRENT TEMPORAL SYSTEM_TIME or CURRENT TEMPORAL BUSINESS_TIME has a value other than the null value.
51047	The activation group of the program or service program referenced by an external procedure is not valid.

Table 44. Class Code 53: Invalid Operand or Inconsistent Specification

SQLSTATE	Meaning
53001	A clause is invalid, because the table space is a workfile.
53004	DSNDB07 is the implicit workfile database.
53014	The specified OBID is invalid.

SQLSTATE	Meaning
53022	Variable or parameter is not allowed.
53035	Key limits must be specified in the CREATE or ALTER INDEX statement.
53036	The number of PARTITION specifications is not the same as the number of partitions.
53037	A partitioned index cannot be created on a table in a non-partitioned table space.
53038	The number of key limit values is zero or greater than the number of columns in the key.
53039	The PARTITION clause of the ALTER statement is omitted or invalid.
53040	The buffer pool cannot be changed as specified.
53041	The page size of the buffer pool is invalid.
53043	Columns with different field procedures cannot be compared.
53044	The columns have a field procedure, but the field types are not compatible.
53045	The data type of the key limit constant is not the same as the data type of the column.
53060	Public dbspaces must be acquired from a recoverable storage pool.
53088	LOCKMAX is inconsistent with the specified LOCKSIZE.
53089	The number of variable parameters for a stored procedure is not equal to the number of expected variable parameters.
53090	Only data from one encoding scheme, either ASCII, EBCDIC or Unicode, can be referenced in the same SQL statement.
53091	The encoding scheme specified is not the same as the encoding scheme currently in use for the containing table space.
53092	Type 1 index cannot be created for a table using the ASCII encoding scheme.
53093	The CCSID ASCII or UNICODE clause is not supported for this database or table space.
53094	The PLAN_TABLE cannot be created with the FOR ASCII clause.
53095	CREATE or ALTER statement cannot define an object with the specified encoding scheme.
53096	The PARTITION clause was specified on CREATE AUXILIARY TABLE, but the base table is not partitioned.
53098	The auxiliary table cannot be created because a column was specified that is not a LOB column.
53099	A WLM ENVIRONMENT name must be specified on the CREATE FUNCTION statement.
530A1	An ALTER TABLE statement specified FLOAT as the new data type for a column, but there is an existing index or constraint that restricts the use of FLOAT.
530A2	The PARTITIONING clause is not allowed on the specified index.
530A3	The specified option is not allowed for the internal representation of the routine specified
530A4	The options specified on ALTER statement are not the same as those specified when the object was created.
530A5	The REGENERATE option is only valid for an index with key expressions.
530A7	EXCHANGE DATA is not allowed because the tables do not have a defined clone relationship.
530A8	A system parameter is incompatible with the specified SQL statement.
530AA	The specified member subset attribute is not valid.
530AB	The member cannot be dropped because it is the only member in the member subset.
530AC	Data in HDFS cannot be mapped to Hadoop table definition.
530AD	Hadoop table definition in BIG SQL Catalog does not match the meta data in the Hive metastore.

Table 45. Class Code 54: SQL or Product Limit Exceeded

SQLSTATE	Meaning
54001	The statement is too long or too complex.
54002	A string constant is too long.
54004	The statement has too many table names or too many items in a SELECT or INSERT list.
54005	The sort key is too long, or has too many columns.
54006	The result string is too long.
54008	The key is too long, a column of the key is too long, or the key many columns.
54009	Too many users were specified in GRANT or REVOKE.
54010	The record length of the table is too long.
54011	Too many columns were specified for a table, view, or table function.
54012	The literal is too long.
54013	The statement has too many host variables.
54014	Too many cursors are open in a unit of work.
54015	A section was not created as a result of executing the null form of an extended dynamic PREPARE, or preprocessing a PREPARE statement.
54016	No more tables can be created in this dbspace.
54017	The maximum number of active packages for a unit of work has been exceeded.
54018	The row is too long.
54019	The maximum number of late descriptors has been exceeded, probably because too many different CCSIDs were used.
54020	No more indexes can be created for this table.
54021	Too many constraints, or the size of the constraint is too large.
54023	The limit for the number of parameters or arguments for a function or a procedure has been exceeded.
54024	The check constraint, generated column, or key expression is too long.
54025	The table description exceeds the maximum size of the object descriptor.
54027	The catalog has the maximum number of user-defined indexes.
54028	The maximum number of concurrent LOB handles has been reached.
54029	The maximum number of open directory scans has been reached.
54030	The maximum number of event monitors are already active.
54031	The maximum number of files have already been assigned the event monitor.
54032	The maximum size of a table has been reached.
54033	The maximum number of partitioning maps has been reached.
54034	The combined length of all container names for the table space is too long.
54035	An internal object limit exceeded.
54036	The path name for the container or storage path is too long.
54037	The container map for the table space is too complicated.
54038	Maximum depth of nested routines or triggers was exceeded.
54039	The container size is too small or too large.
54040	Too many references to transition variables and transition tab columns or the row length for these references is too long.
54041	The maximum number of internal identifiers has been reached.
54042	Only one index is allowed on an auxiliary table.

SQLSTATE	Meaning
54044	A multiple-byte (UCS-2) collating sequence table cannot be supported in DRDA because it is too large.
54045	The maximum level of a type hierarchy has been reached.
54046	The maximum allowable parameters is exceeded in an index extension.
54047	The maximum size of a table space is exceeded.
54048	A temporary table space with sufficient page size does not exist.
54049	Length of an instance of a structured type exceeds the system limit.
54050	The maximum allowable attributes is exceeded in a structured type.
54051	Value specified on FETCH ABSOLUTE or RELATIVE is invalid.
54052	The number of block pages for a buffer pool is too large for the buffer pool.
54053	The value specified for BLOCKSIZE is not in the valid range.
54054	The number of data partitions, or the combination of the number of table space partitions and the corresponding length of the partitioning limit key is exceeded
54055	The maximum number of versions has been reached for a table or index.
54056	A server with the CAT_ENABLE option set to 'YES' already exists.
54057	An XML element name, attribute name, namespace prefix or URI is too long.
54058	The internal representation of an XML path is too long.
54059	A text node string value with only whitespace characters is too long for STRIP WHITESPACE processing.
54061	Too many elements were specified for the security label component.
54062	The maximum number of components in a security policy has been exceeded.
54063	The PCTDEACTIVATE limit has been reached for the event monitor.
54064	More than 65533 instances of a cursor have been opened.
54065	The maximum of 99999 implicitly generated object names has been exceeded.
54066	Recursion limit exceeded within hierarchical query.
54067	The maximum number of connections has been exceeded.
54068	Seamless automatic client reroute retry limit exceeded.

Table 46. Class Code 55: Object Not in Prerequisite State

SQLSTATE	Meaning
55001	The database must be upgraded.
55002	The explanation table is not defined properly.
55003	The DDL registration table is not defined properly.
55004	The database cannot be accessed, because it is no longer a shared database.
55005	Recursion is only supported to a DB2 for i application server.
55006	The object cannot be dropped, because it is currently in use by the same application process.
55007	The object cannot be altered, because it is currently in use by the same application process.
55009	The system attempted to write to a read-only file or a write-protected storage medium.
55011	The operation is disallowed, because the workfile database is not in the stopped state.
55012	A clustering index is not valid on the table.
55014	The table does not have an index to enforce the uniqueness of the primary key.

SQLSTATE	Meaning
55015	The ALTER statement cannot be executed, because the pageset is not in the stopped state.
55016	The ALTER statement is invalid, because the pageset has user-managed data sets.
55017	The table cannot be created in the table space, because it already contains a table.
55018	The schema cannot be dropped, because it is in the library list.
55019	The object is in an invalid state for the operation.
55020	A work file database is already defined for the member.
55021	Change of data type or length of host variable is invalid, because blocking is in effect.
55022	The file server is not registered with this database.
55023	An error occurred calling a procedure.
55024	The table space cannot be dropped, because data related to a table is also in another table space.
55025	The database must be restarted.
55026	A temporary table space cannot be dropped.
55027	The current unit of work is only prepared to process a COMMIT or ROLLBACK statement.
55028	Parameter in the LASTING GLOBALV file is either missing or incorrect.
55029	Local program attempted to connect to a remote database.
55030	A package specified in a remote BIND REPLACE operation must not have a system list.
55031	The format of the error mapping file is incorrect.
55032	The CONNECT statement is invalid, because the database manager was stopped after this application was started.
55033	An event monitor or usage list cannot be activated in the same unit of work in which it is created or modified.
55034	The event monitor is in an invalid state for the operation.
55035	The table cannot be dropped, because it is protected.
55037	The distribution key cannot be dropped because the table is in a multi-partition database partition group.
55038	The database partition group cannot be used because it is being rebalanced.
55039	The access or state transition is not allowed, because the table space is not in an appropriate state.
55040	The database's split image is in the suspended state.
55041	Containers cannot be added to a table space while a rebalance is in progress.
55042	The alias is not allowed because it identifies a single member of a multiple member file.
55043	The attributes of a structured type cannot be altered when a typed table of typed view based on the type exists.
55044	The PROCEDURE must have a status of STOP-REJ, or the PSERVER must be stopped with IMPL=N, before it can be altered or dropped.
55045	The SQL Archive (SAR) file for the routine cannot be created because a necessary component is not available at the server.
55046	The specified SQL archive (SAR) does not match the target environment.
55047	A routine declared as NOT FEDERATED attempted to access a federated object.
55048	Encrypted data cannot be encrypted.
55049	The event monitor table is not properly defined.
55050	An object cannot be created into a protected schema.
55051	The ALTER BUFFER POOL statement is currently in progress.

SQLSTATE	Meaning
55054	A method cannot be called recursively.
55055	INSERT, UPDATE or DELETE failed on the continuously available table because of a continuous availability failure.
55056	The nickname statistics cannot be updated because the database is not enabled for federation.
55057	The statement or command is not allowed while the table has detached dependents or until the asynchronous partition detach task completes.
55058	The DEBUG MODE cannot be changed for a routine that was created with DISABLE DEBUG MODE.
55059	The currently active version for a routine cannot be dropped.
55060	Automatic storage has not been defined for the database.
55061	Table space storage cannot be changed for an automatic storage table space.
55062	Storage paths cannot be provided because the database is not enabled for automatic storage.
55063	The XML schema is not in the correct state for the operation.
55064	Label-based access control cannot be applied to the column because the table has no security policy.
55065	Label-based access control cannot be applied to the column because the table has no security policy.
55067	The table cannot be protected because a materialized query table or a staging table depends on it.
55068	A row change timestamp expression cannot be used because the table does not have a row change timestamp.
55069	Creating or invoking a sourced procedure using a wrapper defined as fenced is not supported.
55070	The administration task table table is not properly defined.
55071	The request cannot be performed because a database partition is being added.
55072	A database partition cannot be added because an incompatible command is already in progress.
55073	Request failed because a storage path is in the drop pending state.
55074	The explain facility failed because the specified activity event monitor is not a write-to-table event monitor.
55075	The explain facility is not supported for the specified section.
55076	Federation is not supported for a table with XML data when the Database Partitioning Feature is enabled.
55077	The operation on the database partition group cannot be performed until all applications in the instance are aware of the new database partition server.
55078	The table is already in the specified state.
55079	The operation cannot be performed becuase the XML column is not in the versioning format.
5507A	The analytics routine could not be retrieved for ANALYZE_TABLE processing.

Table 47. Class Code 56: Miscellaneous SQL or Product Error

SQLSTATE	Meaning
56004	The statement failed, because the Invalid Entities table is full.
56010	The subtype of a string variable is not the same as the subtype at bind time, and the difference cannot be resolved by character conversion.
56016	The ranges specified for data partitions are not valid.
56018	A column cannot be added to the table, because it has an edit procedure.

SQLSTATE	Meaning
56023	An invalid reference to a remote object has been detected.
56025	An invalid use of AT ALL LOCATIONS in GRANT or REVOKE has been detected.
56027	A nullable column of a foreign key with a delete rule of SET NULL cannot be part of the key of a partitioned index.
56031	The clause or scalar function is invalid, because mixed and DBCS data are not supported on this system.
56033	The insert or update value of a long string column must be a variable or NULL.
56034	ALLUSERS can only be used in GRANT CONNECT without a password.
56035	Referential constraints cannot cross dbspaces resident in different types of storage pools.
56036	Specific and non-specific volume IDs are not allowed in a storage group.
56038	The requested feature is not supported in this environment.
56040	CURRENT SQLID cannot be used in a statement that references remote objects.
56041	An Extended PREPARE can only be executed using the DRDA protocol if it has an input SQLDA.
56042	Only one package can be created or modified in a unit of work, and, while that package is being created or modified, all statements in that unit of work must be issued against that package. If the package is non-modifiable, only Extended PREPARE statements can be issued.
56044	An attempt was made to execute a section that has been marked invalid in a modifiable package that is undergoing modification.
56045	The application must issue a rollback operation to back out the change that was made at the read-only application server.
56046	CREATE PACKAGE with the REPLACE option cannot be issued against a modifiable package.
56047	PREPARE Adding Empty Section was not preceded by a CREATE PACKAGE with the NOMODIFY option.
56048	Three-part package names are not supported.
56049	An unexpected error occurred when attempting to rebind a view with a new version of the database manager. The view must be dropped and recreated.
56052	The remote requester tried to bind, rebind, or free a trigger package.
56053	The parent of a table in a read-only shared database must also be a table in a read-only shared database.
56054	User-defined datasets for objects in a shared database must be defined with SHAREOPTIONS(1,3).
56055	The database is defined as SHARE READ, but the table space or indexspace has not been defined on the owning system.
56056	The description of an object in a SHARE READ database must be consistent with its description in the OWNER system.
56057	A database cannot be altered from SHARE READ to SHARE OWNER.
56058	A COMMIT WORK statement or a ROLLBACK WORK statement cannot be dynamically prepared or executed.
56059	An error occurred when binding a triggered SQL statement.
56060	An LE function failed.
56062	A distributed operation is invalid, because the unit of work was started before DDF.
56063	In Single User Mode only one CICS task can issue an SQL statement.
56064	The bind operation is disallowed, because the program depends on functions of a release from which fallback has occurred.
56065	The bind operation is disallowed, because the DBRM has been modified or was created for a different release.

SQLSTATE	Meaning
56066	The rebind operation is disallowed, because the plan or package depends on functions of a release from which fallback has occurred.
56067	The rebind operation is disallowed, because the value of SYSPACKAGE.IBMREQD is invalid.
56072	Execution failed due to the function not supported by a downlevel server that will not affect the execution of subsequent SQL statements.
56073	Execution failed due to the function not supported by a downlevel server that will affect the execution of subsequent SQL statements.
56076	A DB2 Server for VSE or VM application requestor that uses DRDA-only protocols cannot be connected to a DB2 Server for VSE or VM application server that uses SQLDS-only protocols.
56079	Neither protocol option AUTO nor DRDA can be specified, because the DRDA facility has not been installed for the application requester.
56080	The data type is not allowed in DB2 private protocol processing.
56084	An unsupported SQLTYPE was encountered in a select list or input list.
56088	ALTER FUNCTION failed because functions cannot modify data when they are processed in parallel.
56089	Specified option requires type 2 indexes.
56090	The alter of an index or table is not allowed.
56091	Multiple errors occurred as a result of executing a compound SQL statement.
56092	The type of authorization cannot be determined, because the authorization name is both a user id and group id.
56093	A query includes a column with a data type not supported by the application requestor.
56095	A bind option is invalid.
56096	Bind options are incompatible.
56097	LONG VARCHAR and LONG VARGRAPHIC column are not permitted in table spaces using DEVICES.
56098	An error occurred during an implicit rebind, recompile, or revalidation.
56099	The REAL data type is not supported by the target database.
560A0	Action on a LOB value failed.
560A1	The table space name is not valid.
560A2	A LOB table and its associated base table space must be in the same database.
560A3	The table is not compatible with the database.
560A4	The operation is not allowed on an auxiliary table.
560A5	An auxiliary table already exists for the specified column or partition.
560A6	A table cannot have a LOB column unless it also has a ROWID column or cannot have an XML column unless it also has a DOCID.
560A7	GBPCACHE NONE cannot be specified for a table space or index in GRECP.
560A8	An 8K or 16K buffer pool pagesize is invalid for a WORKFILE object.
560A9	A discontinued parameter, option, or clause was specified.
560AA	The use of this clause or scalar function is supported only for Unicode database graphic data.
560AB	The data type is not supported in an SQL routine.
560AC	Wrapper definition cannot be used for the specified type or version of data source.
560AD	A view name was specified after LIKE in addition to the INCLUDING IDENTITY COLUMN ATTRIBUTES clause.
560AE	The specified table or view is not allowed in a LIKE clause.

SQLSTATE	Meaning
560AF	Prepare statement is not supported when using gateway concentrator.
560B0	Invalid new size value for table space or table space container resizing.
560B1	Procedure failed because a result set was scrollable but the cursor was not positioned before the first row.
560B2	Open failed because the cursor is scrollable but the client does not support scrollable cursors.
560B3	Procedure failed because one or more result sets returned by the procedure are scrollable but the client does not support scrollable cursors.
560C0	The table is defined as CCSID UNICODE and cannot be used in an SQL function or SQL method.
560C1	Tables created in the Unicode encoding scheme cannot be a typed table, or contain graphic types or user-defined types.
560C2	Writing a history file entry for a dropped table failed.
560C3	An AFTER trigger cannot modify a row being inserted for an INSERT statement.
560C4	The option is not valid for the ARD interface.
560C5	The package must be bound or rebound to be successfully executed.
560C6	A referential constraint cannot modify a row that was modified by an SQL data change statement within a fullselect.
560C7	ALTER VIEW failed.
560C8	Some of the nickname statistics cannot be updated.
560C9	An error occurred while explaining a reoptimizable statement.
560B5	Local special register is not valid as used.
560B7	For a multiple-row INSERT, the usage of a sequence expression must be the same for each row.
560B8	The SQL statement cannot be executed because it was precompiled at a level that is incompatible with the current value of the ENCODING bind option or special register.
560B9	Hexadecimal constant GX is not allowed.
560BB	The same variable must be used in both the USING and INTO clauses for an INOUT parameter in a dynamically prepared CALL statement.
560BC	An error has occurred when accessing the configuration file.
560BD	Unexpected error code received from data source.
560BE	Operation prohibited on a continuously available table.
560BF	The encryption and decryption facility has not been installed.
560C0	The table is defined as CCSID UNICODE and cannot be used in an SQL function or SQL method.
560C1	Tables created in the Unicode encoding scheme cannot be a typed table, or contain graphic types or user-defined types.
560C2	Writing a history file entry for a dropped table failed.
560C3	An AFTER trigger cannot modify a row being inserted for an INSERT statement.
560C4	The option is not valid for the ARD interface.
560C5	The package must be bound or rebound to be successfully executed.
560C6	A referential constraint cannot modify a row that was modified by an SQL data change statement within a fullselect.
560C7	ALTER VIEW failed.
560C8	Some of the nickname statistics cannot be updated.
560C9	An error occurred while explaining a reoptimizable statement.
560CA	The SQL statement references a routine which can only be run on the current database partition.

SQLSTATE	Meaning
560CB	A federated server received a SOAP Fault from a web services data source.
560CC	ALTER INDEX failed.
560CE	An SQL variable has been invalidated by a COMMIT or ROLLBACK operation.
560CG	An XML value contains a combination of XML nodes that causes an internal identifier limit to be exceeded.
560CH	The maximum number of children nodes for an XML node in an XML value is exceeded.to be exceeded.
560CI	The result set specified to be returned to the client is invalid.
560CJ	The table space must be created in the IBMCATGROUP database partition group.
560CK	Explain monitored statements failed.
560CL	Creating or altering the sourced procedure is not supported at this data source.
560CM	An error occurred in a key expression evaluation.
560CN	The wrapper is not compatible with the release of DB2 installed at the federated server.
560CO	Cycle detected in a hierarchical query.
560CP	Both DEFAULT and explicit values cannot be specified for a column defined as ROW CHANGE TIMESTAMP and GENERATED BY DEFAULT.
560CR	The XML Toolkit LPO, the Java JDK or JVM, or PASE is not installed.
560CS	The event monitor may not have started or may not have started with full restart capability.
560CT	The module alias cannot be used to specify the module name as the target module for the DDL statement.
560CU	The VARCHAR option is not consistent with the option specified when the procedure was created.
560CV	Invalid table reference for table locator.
560CW	Operation cannot be performed on a host where a DB2 PowerHA pureScale server, also known as CF, is located.
560CX	A table that has a trigger or is a parent of a referential constraint with a delete rule of CASCADE, SET NULL, or SET DEFAULT is not allowed as the target table in a MERGE statement that contains a global variable, function, or subselect in an assignment or values clause.
560CY	A period specification or period clause is not valid as specified.
560CZ	A discontinued command, API function, or SQL statement was specified.
560D0	An invalid object cannot be implicitly revalidated.
560D1	MERGE not allowed because operations performed in the MERGE may affect other operations in the MERGE.
560D2	An autonomous procedure was terminated abnormally.
560D3	SQL statement uses a clause that cannot be processed by the federated data source.
560D4	An in-database analytics provider returned an unexpected error.
560D5	The statement cannot be executed by the query accelerator.
560D6	Package failed to rebind because it is generated for a compiled SQL object that is invalid.
560D8	The operation failed because a valid license key was not found.
560D9	Invalid value for a property in a configuration file.

Table 48. Class Code 57: Resource Not Available or Operator Intervention

SQLSTATE	Meaning
57001	The table is unavailable, because it does not have a primary index.

SQLSTATE	Meaning
57002	GRANT and REVOKE are invalid, because authorization has been disabled.
57003	The specified buffer pool has not been activated.
57004	The table is unavailable, because it lacks a partitioned index.
57005	The statement cannot be executed, because a utility or a governor time limit was exceeded.
57006	The object cannot be created, because a DROP or CREATE is pending.
57007	The object cannot be used, because an operation is pending.
57008	The date or time local format exit has not been installed.
57009	Virtual storage or database resource is temporarily unavailable.
57010	A field procedure could not be loaded.
57011	Virtual storage or database resource is not available.
57012	A non-database resource is not available. This will not affect the successful execution of subsequent statements.
57013	A non-database resource is not available. This will affect the successful execution of subsequent statements.
57014	Processing was canceled as requested.
57015	Connection to the local DB2 not established.
57016	The table cannot be accessed, because it is inactive.
57017	Character conversion is not defined.
57018	A DDL registration table or its unique index does not exist.
57019	The statement was not successful, because of a problem with a resource.
57020	The drive containing the database is locked.
57021	The specified I/O device is not ready.
57022	The table could not be created, because the authorization ID of the statement does not own any suitable dbspaces.
57023	The DDL statement cannot be executed, because a DROP is pending of a DDL registration table.
57024	No appropriate CMS message repository can be accessed.
57025	There is not enough room in the dbspace(s) allocated to hold packages.
57026	The system dbspace SYS002 does not exist. This dbspace is used to store packages.
57027	The connection to the application server has been severed by the operator.
57028	The unit of work has been rolled back due to an excessive number of system wide lock requests.
57029	The unit of work has been rolled back due to an excessive number of lock requests by the unit of work.
57030	Connection to application server would exceed the installation-defined limit.
57031	Connection to the application server is not possible, because the DB2 Server for VSE or VMS virtual machine does not have access to that application server.
57032	The maximum number of concurrent databases have already been started.
57033	Deadlock or timeout occurred without automatic rollback.
57036	The transaction log does not belong to the current database.
57037	The ACQUIRE DBSPACE statement failed, because all storage pools for available dbspaces are full.
57038	No space is available in the storage pool.
57039	The VSE Online Resource Manager has been shut down, either by the operator, or due to a serious error.

SQLSTATE	Meaning
57040	The communications directory was either not found, or it has the wrong file type.
57042	DDM recursion has occurred.
57043	A local SQL application program cannot be executed on an application server.
57044	The resource adapter cannot find an entry for the character set in the ASISSCR MACRO file.
57045	The resource adapter cannot find an entry for the character set in the SYSCHARSETS file.
57046	A new transaction cannot start because the database or instance is quiesced.
57047	An internal database file cannot be created, because the directory is not accessible.
57048	An error occurred while accessing a table space.
57049	The operating system process limit has been reached.
57050	The file server is not currently available.
57051	The estimated CPU cost exceeds the resource limit.
57052	The database partition is unavailable because it does not have containers for all temporary table spaces.
57053	A table is not available in a routine or trigger because of violated nested SQL statement rules.
57054	A table is not available until the auxiliary tables and indexes for its externally stored columns have been created.
57055	A temporary table space with sufficient page size was not available.
57056	The package is not available because the database is in NO PACKAGE LOCK mode.
57057	The SQL statement cannot be executed due to a prior condition in a DRDA chain of SQL statements.
57059	There is not enough space in the table space for the specified action.
57060	The statement cannot be processed because no transport from the transport pool is available.
57061	The current state of a member prevents processing of the statement.
57062	Adjustment not allowed for a period as a result of a data change operation.
57063	The current member cannot process data change statements because of an error on another member.
57064	Multiple active statements are not supported on the connection to the federated data source.
57065	In-database analytics provider is not available for communication with the database manager.
57066	The statement was not successful because of a communication error with a Big SQL component.
57067	A connection to a Hadoop I/O component could be established or maintained.source.

Table 49. Class Code 58: System Error

SQLSTATE	Meaning
58001	The database cannot be created, because the assigned DBID is a duplicate.
58002	An exit has returned an error or invalid data.
58003	An invalid section number was detected.
58004	A system error (that does not necessarily preclude the successful execution of subsequent SQL statements) occurred.
58005	A system error (that prevents the successful execution of subsequent SQL statements) occurred.
58006	A system error occurred during connection.
58007	A system error occurred with datalink file management.

SQLSTATE	Meaning
58008	Execution failed due to a distribution protocol error that will not affect the successful execution of subsequent DDM commands or SQL statements.
58009	Execution failed due to a distribution protocol error that caused deallocation of the conversation.
58010	Execution failed due to a distribution protocol error that will affect the successful execution of subsequent DDM commands or SQL statements.
58011	The DDM command is invalid while the bind process in progress.
58012	The bind process with the specified package name and consistency token is not active.
58013	The SQLCODE is inconsistent with the reply message.
58014	The DDM command is not supported.
58015	The DDM object is not supported.
58016	The DDM parameter is not supported.
58017	The DDM parameter value is not supported.
58018	The DDM reply message is not supported.
58021	A system error occurred while loading a program.
58023	A system error has caused the current program to be canceled.
58024	An error has occurred in the underlying operating system.
58025	A column in a catalog table has the wrong data type.
58026	The number of variables in the statement is not equal to the number of variables in SQLSTTVRB.
58027	The package was not created and unit of work was rolled back due to an earlier system error.
58028	The commit operation failed, because a resource in the unit of work was not able to commit its resources.
58029	An internal error has occurred while attempting to log user data.
58030	An I/O error has occurred.
58031	The connection was unsuccessful, because of a system error.
58032	Unable to use the process for a fenced mode user-defined function.
58033	An unexpected error occurred while attempting to access a client driver.
58034	An error was detected while attempting to find pages for an object in a DMS table space.
58035	An error was detected while attempting to free pages for an object in a DMS table space.
58036	The internal table space ID specified does not exist.
58039	A system error occurred with Hive MetaStore management.
58040	A Big SQL component encountered an error.

Table 50. Class Code 5UA: Common Utilities and Tools

SQLSTATE	Meaning
5UA01	The task cannot be removed because it is currently executing.
5UA02	An error occurred while attempting to add a database partition.
5UA03	An error occurred during the explicit revalidation of an object.
5UA04	No alert has been registered previously with the DBMS_ALERT.REGISTER procedure.
5UA05	An invalid filename was specified for a UTL_FILE module routine.
5UA06	An invalid path was specified for a UTL_FILE module routine.
5UA07	An invalid file handle was specified for a UTL_FILE module routine.

SQLSTATE	Meaning
5UA08	An invalid mode was specified for the UTL_FILE.FOPEN function.
5UA09	An invalid maximum line size was specified for the UTL_FILE.FOPEN function.
5UA0A	A read error was encountered by a UTL_FILE module routine.
5UA0B	A write error was encountered by a UTL_FILE module routine.
5UA0C	The UTL_FILE.FREMOVE procedure failed to delete the specified file.
5UA0D	The UTL_FILE.FRENAME procedure failed to rename the specified file.
5UA0E	A UTL_SMTP module routine encountered a transient SMTP server error.
5UA0F	A UTL_SMTP module routine encountered a permanent SMTP server error.
5UA0G	TCP timeout has occurred.
5UA0H	TCP/IP network error.
5UA0I	The data type, length, scale, value, or CCSID is invalid for a UTL_TCP module routine.
5UA0J	The data type, length, scale, value, or CCSID is invalid for a DBMS_LOB module routine.
5UA0K	Access was denied to a file when executing a UTL_FILE module routine.
5UA0L	An internal error, out of memory or system error, occurred in the UTL_FILE module.
5UA0M	The ADMIN_MOVE_TABLE procedure has terminated with an error.
5UA0N	The operation is invalid because the UTL_SMTP module routine is called out of sequence.
5UA0O	The argument to the WRAP function or to the CREATE_WRAPPED procedure is not valid.
5UA0P	Message buffer size exceeded.

Chapter 14. CCSID values

The following tables describe the CCSIDs and conversions provided by the IBM relational database products. For more information, see “Character conversion” on page 31.

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 77. Universal Character Set (UTF-8, UTF-16, and UCS-2)

CCSID	Description	z/OS	IBM 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 for LUW, 13488 is only used to tag the GRAPHIC column of eucJP and eucTW databases.

Table 78. CCSIDs for EBCDIC Group 1 (Latin-1) Countries

CCSID	Description	z/OS	IBM 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 79. CCSIDs for PC-Data and ISO Group 1 (Latin-1) Countries

CCSID	Description	z/OS	IBM i	AIX	HP	Sun	NT	SCO	SGI	Linux
437	USA	X	C	C	C	C	C	C	C	C
819	Latin-1 countries (ISO 8859-1)	X	C	X	X	X	C	X	X	X
850	Latin Alphabet Number 1; Latin-1 countries	X	C	X	C	C	C	C	C	C
858	Latin Alphabet Number 1; Latin-1 countries (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 80. CCSIDs for EBCDIC Group 1a (Non-Latin-1 SBCS) Countries

CCSID	Description	z/OS	IBM i	AIX	HP	Sun	NT	SCO	SGI	Linux
420	Arabic (Type 4)Visual LTR	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)			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							
1175	Turkey Latin-5 (with Turkish lira)		X							
4971	Greek (with Euro)	X	X							
5233	Devanagari (with Indian rupee)		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 80. CCSIDs for EBCDIC Group 1a (Non-Latin-1 SBCS) Countries (continued)

CCSID	Description	z/OS	IBM 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 81. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries

CCSID	Description	z/OS	IBM 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
1251	Windows Cyrillic	X	C	C	C	C	X	C	C	C

Table 81. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries (continued)

CCSID	Description	z/OS	IBM i	AIX	HP	Sun	NT	SCO	SGI	Linux
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	C							
5347	Windows Cyrillic (with Euro)	X	C							
5349	Windows Greek (with Euro)	X	C							
5350	Windows Turkey (with Euro)	X	C							
5351	Windows Hebrew (with Euro)	X	C							
5352	Windows Arabic (with Euro)	X	C							
5353	Windows Baltic Rim (with Euro)	X	C							
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
62226	Arabic (Type 6)			X	C	C	C	C	C	C

Table 81. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries (continued)

CCSID	Description	z/OS	IBM i	AIX	HP	Sun	NT	SCO	SGI	Linux
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
62237	Hebrew (Type 8)									
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 82. SBCS CCSIDs for EBCDIC Group 2 (DBCS) Countries

CCSID	Description	z/OS	IBM 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 (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 83. SBCS CCSIDs for PC-Data Group 2 (DBCS) Countries

CCSID	Description	z/OS	IBM 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 84. DBCS CCSIDs for EBCDIC Group 2 (DBCS) Countries

CCSID	Description	z/OS	IBM 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 85. DBCS CCSIDs for PC-Data Group 2 (DBCS) Countries

CCSID	Description	z/OS	IBM 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	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 86. Mixed CCSIDs for EBCDIC Group 2 (DBCS) Countries

CCSID	Description	z/OS	IBM 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

Table 86. Mixed CCSIDs for EBCDIC Group 2 (DBCS) Countries (continued)

CCSID	Description	z/OS	IBM i	AIX	HP	Sun	NT	SCO	SGI	Linux
1390	Japan Katakana/Kanji (extended with Euro) - including 4370 UDC	X		C	C	C	C	C	C	C
1399	Japan (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 87. Mixed CCSIDs for PC-Data Group 2 (DBCS) Countries

CCSID	Description	z/OS	IBM 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							
17354	Korea 2022 TCP		C							

Table 87. Mixed CCSIDs for PC-Data Group 2 (DBCS) Countries (continued)

CCSID	Description	z/OS	IBM i	AIX	HP	Sun	NT	SCO	SGI	Linux
25546	Korea 2022 TCP		C							
33722	Japan EUC		C							

Chapter 15. Coding SQL statements in C applications

This section describes the programming techniques that are unique to coding SQL statements within a C program. Throughout this book, C is used to represent either C or C++, except where explicitly noted otherwise.

Defining the SQL communications area in C

A C program that contains SQL statements must include one or both of the following:

- An SQLSTATE variable[182] declared as `char SQLSTATE[6]`
- An SQLCODE variable[182] declared as `long SQLCODE`

or,

- An SQLCA (which contains an SQLSTATE and SQLCODE variable).

The SQLSTATE and SQLCODE values are set by the database manager after each SQL statement is executed. An application can check the SQLSTATE or SQLCODE value to determine whether the last SQL statement was successful. See Chapter 13, "SQLSTATE values—common return codes," on page 997 for more information.

The SQLCA can be coded in a C program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLCA;
```

A standard declaration includes both a structure definition and a static data area named 'sqlca'. The SQLCA must not be defined within an SQL declare section. See Chapter 11, "SQLCA (SQL communication area)," on page 981 and "INCLUDE" on page 808 for more information.

The SQLSTATE, SQLCODE, and SQLCA variables must appear before any executable statements. The scope of the declaration must include the scope of all SQL statements in the program.

Note: Many SQL error messages contain message data that is of varying length. The lengths of these data fields are embedded in the value of the SQLCA sqlerrmc field. Because of these lengths, printing the value of sqlerrmc from a C program might give unpredictable results.

Defining SQL descriptor areas in C

The following statements require an SQLDA:

EXECUTE...USING DESCRIPTOR *descriptor-name*
FETCH...USING DESCRIPTOR *descriptor-name*
OPEN...USING DESCRIPTOR *descriptor-name*
DESCRIBE *statement-name* INTO *descriptor-name*
PREPARE *statement-name* INTO *descriptor-name*

182. In DB2 for z/OS, the STDSQL(YES) option must be in effect to declare the SQLSTATE and SQLCODE variables. In DB2 for LUW, the LANGLEVEL SQL92E option must be used to declare the SQLSTATE and SQLCODE variables.

CALL...USING DESCRIPTOR *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. An SQLDA can be coded in a C program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLDA;
```

A standard declaration includes only a structure definition with the name 'sqlda'. The SQLDA must not be defined within an SQL declare section. See Chapter 12, “SQLDA (SQL descriptor area),” on page 985 and “INCLUDE” on page 808 for more information.

One benefit from using the INCLUDE SQLDA SQL statement is the following macro definition:

```
#define SQLDASIZE(n) (sizeof(struct sqlda) + (n-1)* sizeof(struc sqlvar))
```

This macro makes it easy to allocate storage for an SQLDA with a specified number of SQLVAR elements. In the following example, the SQLDASIZE macro is used to allocate storage for an SQLDA with 20 SQLVAR elements.

```
#include <stdlib.h>
EXEC SQL INCLUDE SQLDA;
struct sqlda *mydaptr;
short numvars = 20;
  .
  .
mydaptr = (struct sqlda *) malloc(SQLDASIZE(numvars));
mydaptr->sqln = 20;
```

Here are other macro definitions that are included with the INCLUDE SQLDA statement:

GETSQLDOUBLED(daptr)
Returns 1 if the SQLDA pointed to by daptr has been doubled, or 0 if it has not been doubled. The SQLDA is doubled if the seventh byte in the SQLDAID field is set to '2'.

SETSQLDOUBLED(daptr, newvalue)
Sets the seventh byte of SQLDAID to newvalue.

GETSQLDALONGLEN(daptr,n)
Returns the length attribute of the *n*th entry in the SQLDA to which daptr points. Use this only if the SQLDA was doubled and the *n*th SQLVAR entry has a LOB data type.

SETSQLDALONGLEN(daptr,n,len)
Sets the SQLLONGLEN field of the SQLDA to which daptr points to len for the *n*th entry. Use this only if the SQLDA was doubled and the *n*th SQLVAR entry has a LOB data type.

GETSQLDALENPTR(daptr,n)
Returns a pointer to the actual length of the data for the *n*th entry in the SQLDA to which daptr points. The SQLDATALEN pointer field returns a pointer to a long (4 byte) integer. If the SQLDATALEN pointer is zero, a NULL pointer is returned. Use this only if the SQLDA has been doubled.

SETSQLDALENPTR(daptr,n,ptr)
Sets a pointer to the actual length of the data for the *n*th entry in the SQLDA to which daptr points. Use this only if the SQLDA has been doubled.

SQLDA declarations can appear wherever a structure definition is allowed. Normal C scope rules apply.

Embedding SQL statements in C

SQL statements can be coded in a C program wherever executable statements can appear.

Each SQL statement in a C program must begin with EXEC SQL and end with a semicolon (;). [183] The EXEC SQL keywords must appear all on one line, but the remainder of the statement can appear on the next and subsequent lines.

For example, an UPDATE statement coded in a C program might be coded as follows:

```
EXEC SQL
   UPDATE DEPARTMENT
   SET MGRNO = :MGR_NUM
   WHERE DEPTNO = :INT_DEPT ;
```

Comments

In addition to SQL comments (--), C comments (/* ... */) can be included within embedded SQL statements wherever a blank is allowed, except between the keywords EXEC and SQL. C Comments can span any number of lines but cannot be nested. [184] Single-line comments (starting with //) can be used in a C++ source program but are not permitted anywhere in a C source program.

Continuation for SQL statements

SQL statements can be contained on one or more lines. An SQL statement can be split wherever a blank can appear. A character-string constant or delimited identifier can be continued on the following line using the backslash (\). Identifiers that are not delimited cannot be continued. For graphic-string constants in EBCDIC, see product documentation.

Cursors

The DECLARE CURSOR statement must precede all statements that explicitly refer to the cursor by name.

Including code

SQL statements or C statements can be included by embedding the following SQL statement at the point in the source code where the statements are to be embedded:

```
EXEC SQL INCLUDE name;
```

C #include statements cannot be used to include SQL statements or declarations of C variables that are referenced in SQL statements.

183. In DB2 for z/OS, if the HOST(C(FOLD)) option is specified, SQL keywords and SQL identifiers are folded to uppercase. When the option is not specified, SQL keywords must be specified in uppercase. For either case, host variables are never folded.

184. In DB2 for z/OS, the STDSQL(YES) option must be in effect to use SQL comments.

Margins

SQL statements must be coded in columns 1 through 80. [185]

Names

Any valid C variable name can be used for a host variable, as long as it:

- does not contain DBCS characters
- is less than or equal to 128 characters in length
- does not begin with 'DB2', 'DSN', 'RDI', or 'SQL' in any combination of uppercase or lowercase letters (these names are reserved for the database manager).

Access plan names must not start with 'DSN'. External entry names must not start with 'DSN', 'RDI', or 'SQL'.

For information on the length of a host identifier, see Table 58 on page 960.

NULLs and NULs

C and SQL both use the word null, but for different meanings. The C language has a null character (NUL), a null pointer (NULL), and a null statement (just a semicolon). The C NUL is a single character which compares equal to 0. The C NULL is a special reserved pointer value that does not point to any valid data object. The SQL null value is a special value that is distinct from all nonnull values and denotes the absence of a (nonnull) value.

Statement labels

Executable SQL statements can be preceded with a label, if desired.

Preprocessor considerations

The precompiler does not support C preprocessor directives.

Trigraphs

Some characters from the C character set are not available on all keyboards. These characters can be entered into a C source program using a sequence of three characters called a *trigraph*. Trigraphs are not supported within SQL statement, however, the following trigraph sequences are supported within host variable declarations:

??(left bracket ([)

??) right bracket (])

??< left brace ({)

??> right brace (})

??/ backslash (\)

185. In DB2 for z/OS, a program preparation option must be used to specify margins 1 and 80. If the program preparation option is not specified, the margins will be 1 and 72.

Handling SQL errors and warnings in C

The SQL WHENEVER statement tests the result of every SQL statement within its scope for an error or warning condition. The target for the GOTO clause in a WHENEVER statement must be within the scope of any SQL statements affected by the WHENEVER statement.

The stand-alone SQLSTATE and SQLCODE or information in the SQLCA can also be used in the detection or further handling of error and warning conditions. See Chapter 11, "SQLCA (SQL communication area)," on page 981 for more information.

Using host variables in C

All host variables used in SQL statements must be explicitly declared. A host variable used in an SQL statement must be declared prior to the first use of the host variable in an SQL statement.

The C statements that are used to define the host variables must be preceded by a BEGIN DECLARE SECTION statement and followed by an END DECLARE SECTION statement.

All host variables within an SQL statement must be preceded by a colon (:).

The names of host variables must be unique within the program, even if the host variables are in different blocks or procedures.

An SQL statement that uses a host variable must be within the scope of the statement in which the variable was declared.

Host variables must not be unions, union elements, or pointers. However, a single pointer can be used to reference an SQLDA. Host variables must not be arrays or array elements unless they are used to represent indicator arrays or indicator variables.

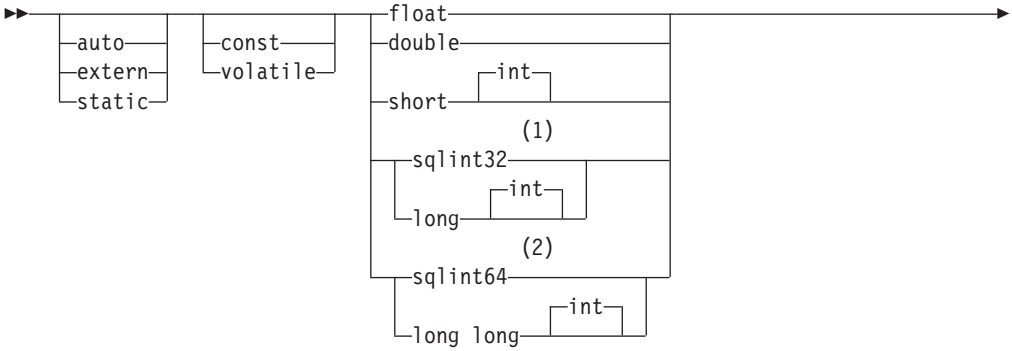
Declaring host variables in C

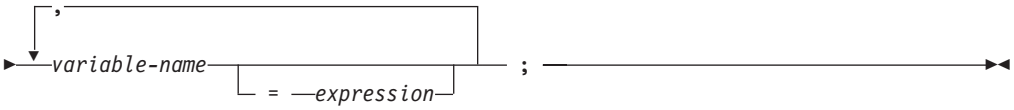
Only a subset of valid C declarations are recognized as valid host variable declarations.

Numeric host variables

The following figure shows the syntax for valid numeric host variable declarations.

Numeric





Notes:

1 For maximum application portability, use sqlint32 for INTEGER host variables. To use sqlint32, the header file sqlsystm.h must be included.

2 For maximum application portability, use sqlint64 for BIGINT host variables. To use sqlint64, the header file sqlsystm.h must be included.

Character host variables (excluding CLOB)

There are three valid non-LOB forms for character host variables:

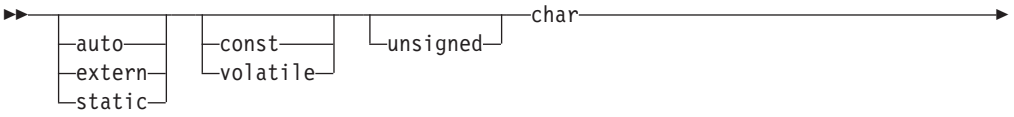
Single-character form

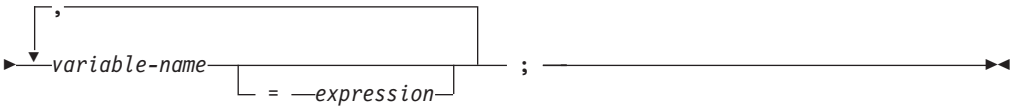
NUL-terminated character form

VARCHAR structured form

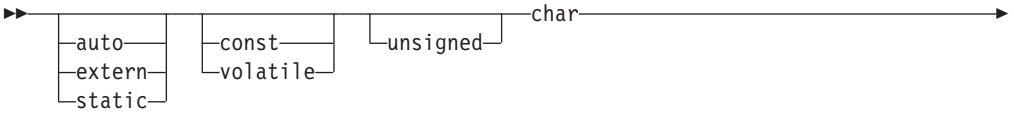
All character types are treated as unsigned.

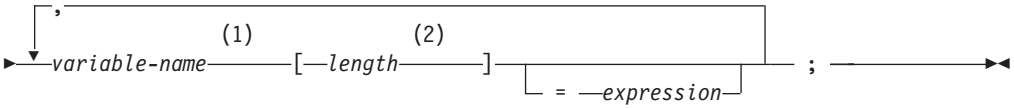
Single-character form





NUL-terminated character form



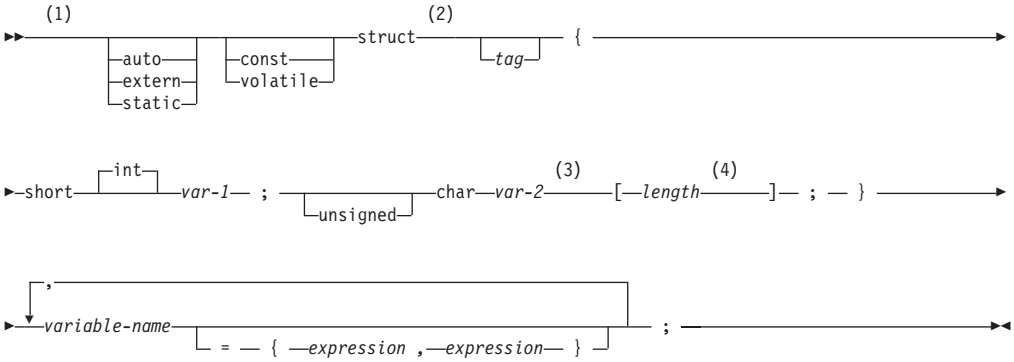


Notes:

1 On input, the string contained by the variable must be NUL-terminated. On output, the string will be NUL-terminated. [186]

2 *length* must be an integer constant greater than 1 and no greater than the maximum length of VARCHAR+1. See Table 60 on page 964 for more information.

VARCHAR structured form



Notes:

1 Use the VARCHAR structured form for bit data that may contain the NULL character. The VARCHAR structured form will not be ended using the NUL-terminator.

2 The struct tag can be used to define other data areas, but these cannot be used as host variables.

3 *var-1* and *var-2* must be simple variable references and cannot be used as host variables.

4 *length* must be an integer constant that is greater than 0 and not greater than the maximum length of VARCHAR. See Table 60 on page 964 for more information.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;

/* valid declaration of host variable vstring */
struct VARCHAR
{
  short len;
  char s[10];
} vstring;

/* invalid declaration of host variable wstring */
struct VARCHAR wstring;
```

Graphic host variables (excluding DBCLOB)

There are three valid non-LOB forms for graphic host variables:

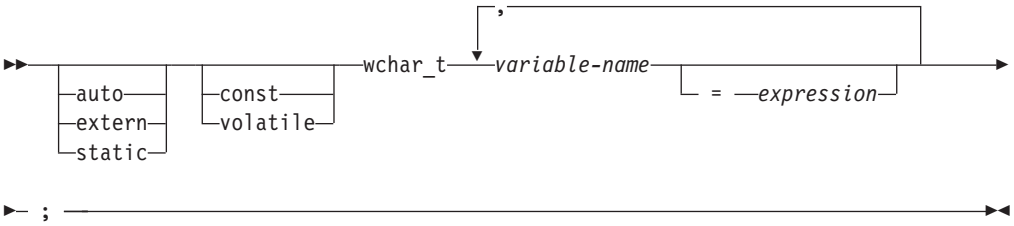
Single-graphic form

NUL-terminated graphic form

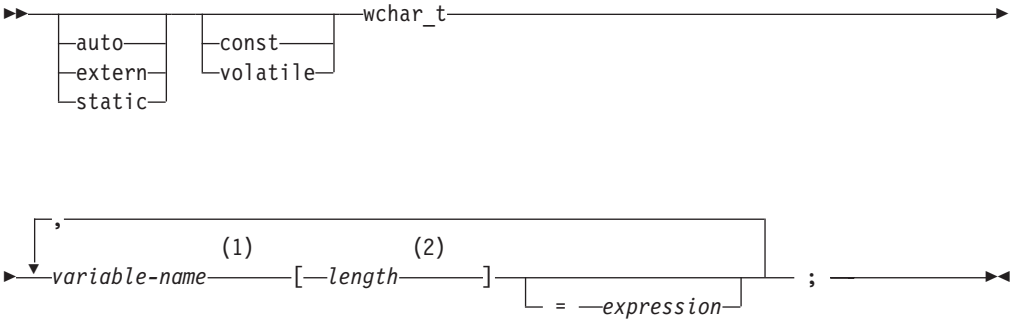
VARGRAPHIC structured form

Single-graphic form

186. In DB2 for i and DB2 for LUW, a program preparation option must be used if the string will be NUL-terminated when the host variable is large enough to contain the result, but not large enough to contain the NUL-terminator. The program preparation option must also be specified for the database manager to verify that NUL-terminated input host variables contain a NUL.



NUL-terminated graphic form

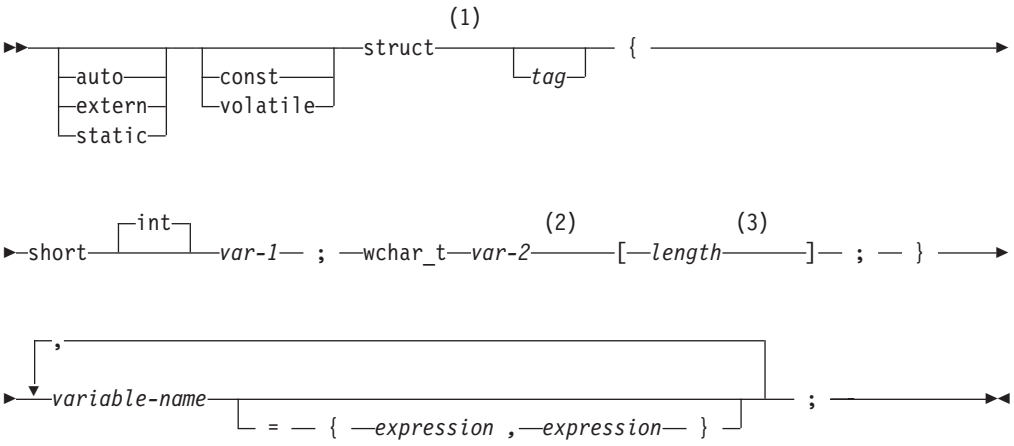


Notes:

1 On input, the string contained by the variable must be NUL-terminated. On output, the string will be NUL-terminated.[186]

2 *length* must be an integer constant that is greater than 1 and not greater than the maximum length of VARGRAPHIC+1. See Table 60 on page 964 for more information.

VARGRAPHIC structured form



Notes:

1 The struct tag can be used to define other data areas, but these cannot be used as host variables.

2 *var-1* and *var-2* must be simple variable references and cannot be used as host variables.

3 *length* must be an integer constant that is greater than 0 and no greater than the maximum length of VARGRAPHIC. See Table 60 on page 964 for more information.

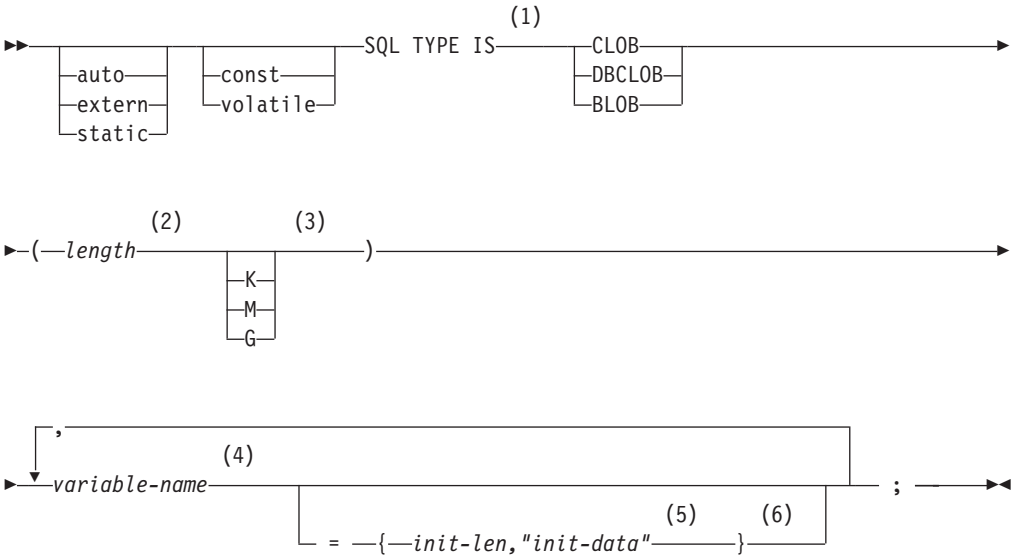
Example:

```
EXEC SQL BEGIN DECLARE SECTION;

/* valid declaration of host variable vstring */
struct VARGRAPH
{
  short len;
  wchar_t s[10];
} vstring;

/* invalid declaration of host variable wstring */
struct VARGRAPH wstring;
```

LOB host variables

C does not have variables that correspond to the SQL data types for LOBs (large objects). To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a C language structure in the output source.

LOB host variable



Notes:

1 SQL TYPE IS, CLOB, DBCLOB, BLOB, K, M, G can be in mixed case.

2 *length* must be an integer constant that is greater than 0 and no greater than the maximum length of CLOB. See Table 60 on page 964 for more information. The maximum value for *length* is further restricted if K, M or G is specified or if DBCLOB is specified.

3 K multiplies *length* by 1024. M multiplies *length* by 1 048 576. G multiplies *length* by 1 073 741 824.

4 The precompiler generates a structure tag which can be used to cast to the host variable's type.

5 The initialization length, *init-len*, must be an integer constant (that is, it

cannot include K, M, or G) that is greater than 0 and not greater than the maximum length of a character constant. See Table 60 on page 964 for more information.

6 If the LOB is not initialized within the declaration, then no initialization will be done within the precompiler generated code. If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an 'L' (indicating a wide-character string).

Examples: *Example 1:* The following declaration:

```
SQL TYPE IS CLOB(128K) var1, var2 = {10, "data2data2"};
```

Results in the effective generation of the following structure:

```
struct var1_t
{
  unsigned long  length;
  char           data[131072];
} var1, var2 = {10, "data2data2"};
```

Example 2: The following declaration:

```
SQL TYPE IS DBCLOB(128K) my_dbclob;
```

Results in the effective generation of the following structure:

```
struct my_dbclob_t {
  unsigned long  length;
  wchar_t        data[131072];
} my_dbclob;
```

Example 3: The following declaration:

```
SQL TYPE IS BLOB(128K) my_blob;
```

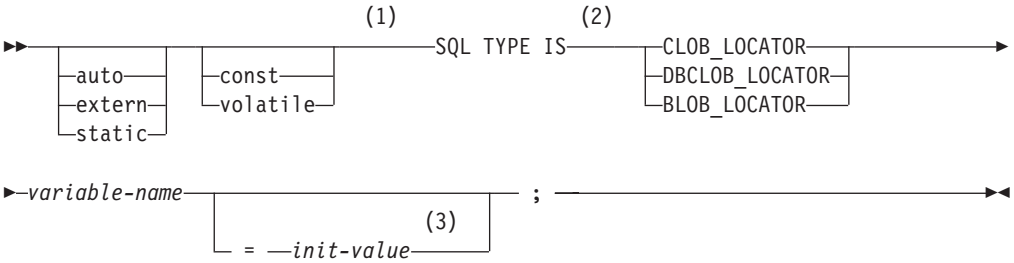
Results in the effective generation of the following structure:

```
struct my_blob_t {
  unsigned long  length;
  char           data[131072];
} my_blob;
```

LOB locator

The following shows the syntax for declaring large object locator host variables in C.

LOB locator



Notes:

1 Pointers to LOB Locators can be declared, with the same rules and restrictions as for pointers to other host variable types.

2 SQL TYPE IS, CLOB_LOCATOR, DBCLOB_LOCATOR, BLOB_LOCATOR can be in mixed case.

3 *init-value* permits the initialization of pointer locator variables. Other types of initialization will have no meaning.

Example: The following declaration:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

Results in the effective generation of the following:

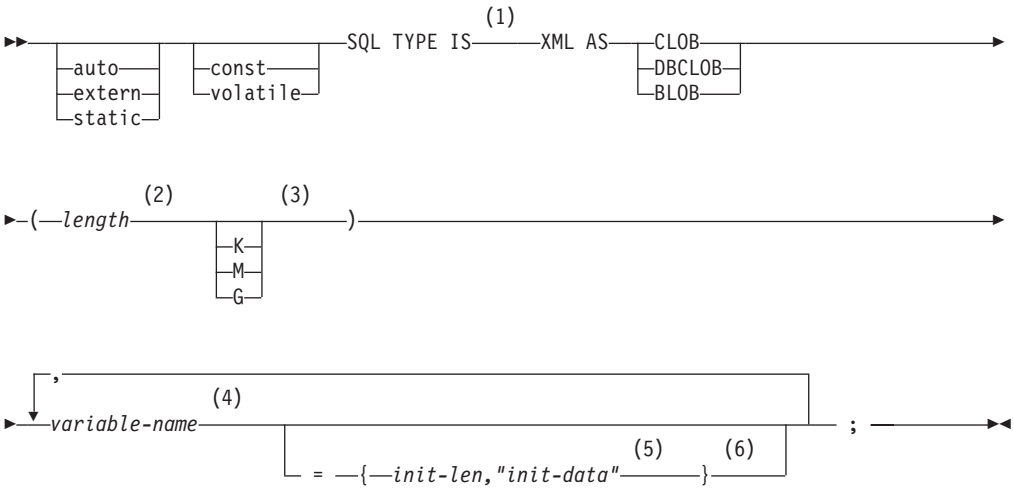
```
unsigned long my_locator;
```

DBCLOB and BLOB locators have similar syntax.

XML host variables

C and C++ do not have variables that correspond to the SQL data type for XML. To create host variables that can be used with this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a C language structure in the output source member.

XML host variable



Notes:

1 SQL TYPE IS, XML AS, CLOB, DBCLOB, BLOB, K, M, G can be in mixed case.

2 *length* must be an integer constant that is greater than 0 and no greater than the maximum length of CLOB. See Table 60 on page 964 for more information. The maximum value for *length* is further restricted if K, M or G is specified or if DBCLOB is specified.

3 K multiplies *length* by 1024. M multiplies *length* by 1 048 576. G multiplies *length* by 1 073 741 824.

4 The precompiler generates a structure tag which can be used to cast to the host variable's type.

5 The initialization length, *init-len*, must be an integer constant (that is, it cannot include K, M, or G) that is greater than 0 and not greater than the maximum length of a character constant. See Table 60 on page 964 for more information.

6 If the XML is not initialized within the declaration, then no initialization will be done within the precompiler generated code. If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an 'L' (indicating a wide-character string).

Example

The following declaration:

```
SQL TYPE IS XML AS CLOB(5000) var1;
```

Results in the effective generation of the following:

```
_Packed struct var1_t {
 unsigned long length;
 char data[5000];
 } var1;
```

Indicator variables in C

An indicator variable is a two-byte integer (short int). On retrieval, an indicator variable is used to show whether its associated host variable has been assigned a null value. On assignment to a column, a negative indicator variable is used to assign a null value.

See “References to host variables” on page 116 for more information on the use of indicator variables.

Indicator variables are declared in the same way as host variables, and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

Example: Given the statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :ClsCd,
                               :Day :DayInd,
                               :Bgn :BgnInd,
                               :End :EndInd;
```

Variables can be declared as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
char  ClsCd[8];
char  Bgn[9];
char  End[9];
short Day, DayInd, BgnInd, EndInd;
EXEC SQL END DECLARE SECTION;
```

Declaring host structures in C

Host structures can be defined in C programs. A host structure contains an ordered group of elementary C variables. It can have a maximum of two levels, even though the host structure might itself occur within a multilevel structure. The one exception is the declaration of a varying-length string, which requires another structure and hence one more level. When the host structure occurs within a multilevel structure, it must be the deepest level of the nested structure. The following is an example of a host structure.

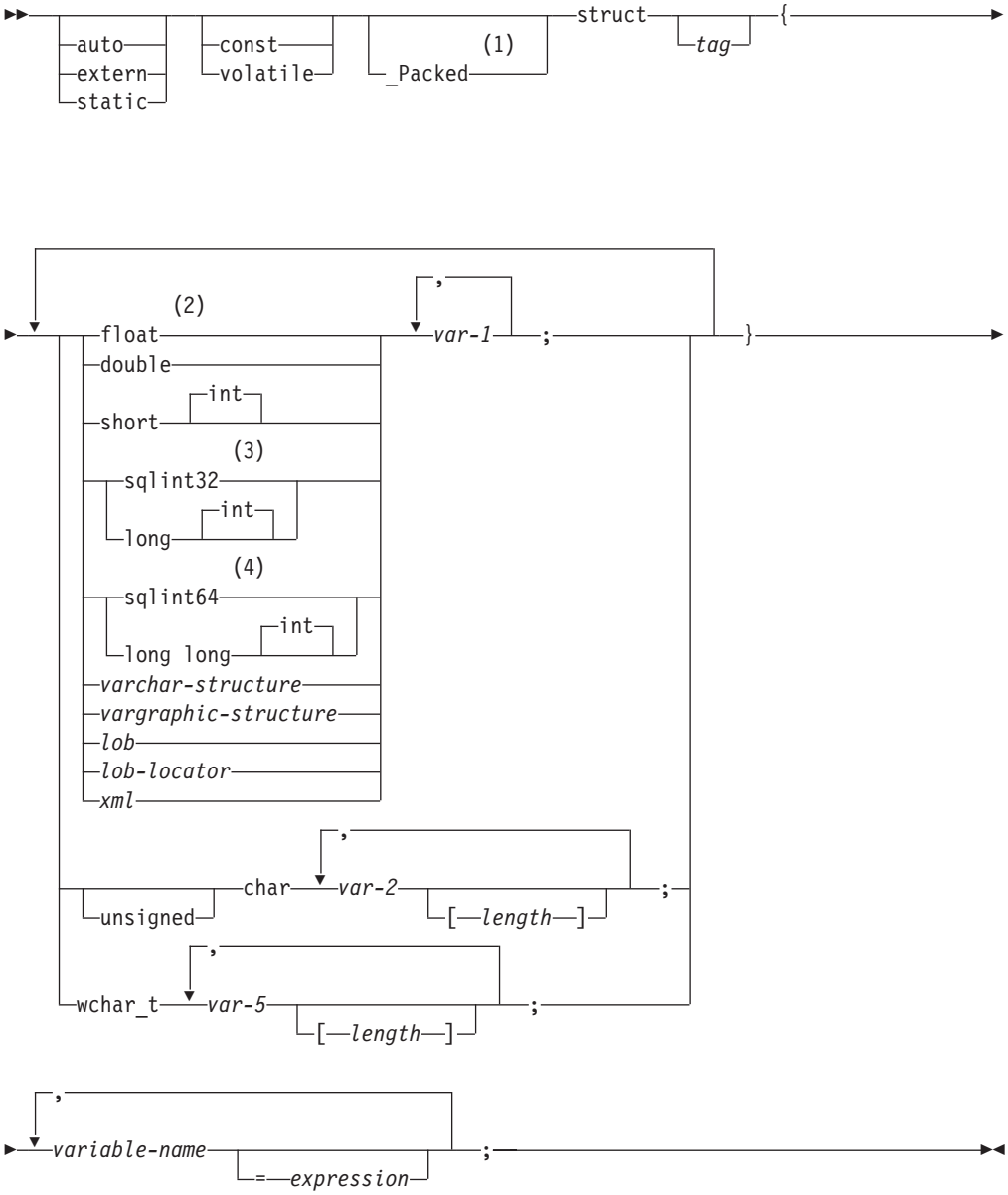
```
struct
{
  char  c1[3];
  struct
  {
    short len;
```

```
    char  data[5];
  } c2;
  char  c3[2];
} target;
```

In this example, *target* is the name of a host structure consisting of the *c1, c2,* and *c3* fields. *c1* and *c3* are character arrays, and *c2* is the host variable equivalent to the VARCHAR structured form.

The following shows the syntax for valid host structures:

Host structures



Notes:

1 _Packed must not be used in C++. Instead, specify #pragma pack(1) prior to the declaration and #pragma pack() after the declaration. See the example below.

2 For details on declaring numeric, character, and graphic host variables, see the notes under numeric, character and graphic host variables.

3 To use sqlint32, the header file sqlsystm.h must be included.

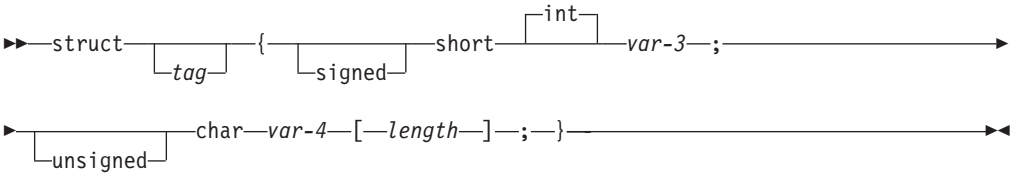
4 To use sqlint64, the header file sqlsystm.h must be included.

Example

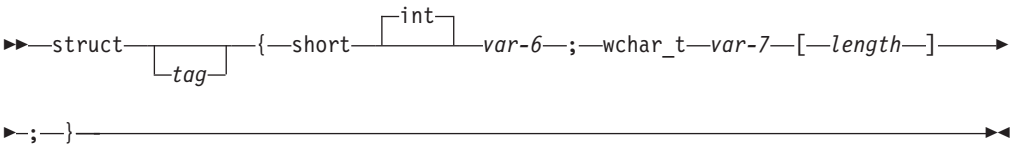
In a C++ program, use pragma to simulate the _Packed option:

```
#pragma pack(1)
struct
{
  short myshort;
  long  mylong;
  char  mychar[5];
} a_st;
#pragma pack()
```

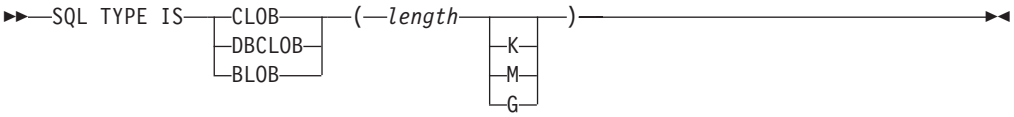
varchar-structure



vargraphic-structure

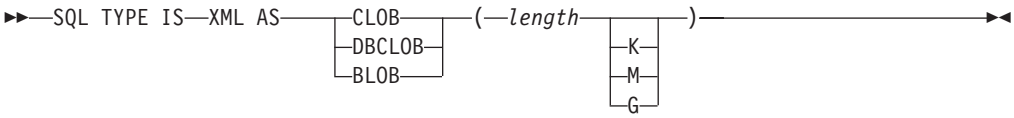


lob



lob-locator

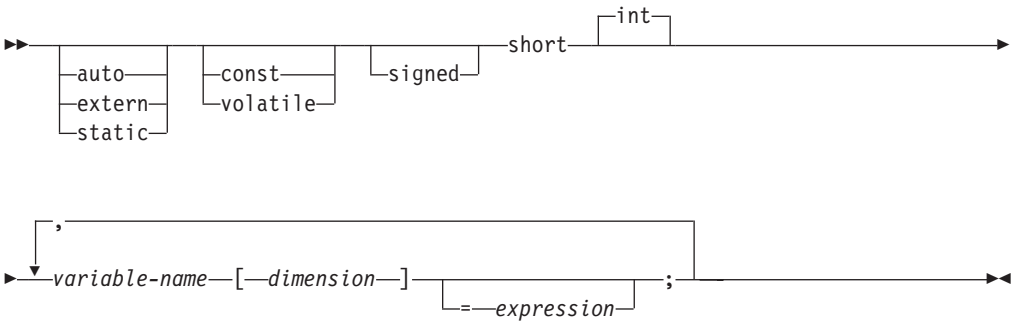
xml



Host structure indicator array

The following figure shows the valid syntax for host structure indicator array declarations.

Host structure indicator array



Note: Dimension must be an integer constant between 1 and 32767.

Using pointer data types in C

A host variable declared in C using pointer notation must be used as a pointer to an SQL descriptor area. A *descriptor-name* can be specified in the CALL, DESCRIBE, EXECUTE, FETCH, OPEN, and PREPARE statements. For example, *descriptor-name* could be declared as:

```
sqlda *outsqlda;
```

and used in a statement as follows:

```
EXEC SQL DESCRIBE STMT1 INTO DESCRIPTOR :*outsqlda;
```

Determining equivalent SQL and C data types

The base SQLTYPE and SQLLEN of host variables are determined according to the following table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.

Table 88. C declarations mapped to typical SQL data types

C Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
short int	500/501	2	SMALLINT
long int	496/497	4	INTEGER
long long int	492/493	8	BIGINT
float	480/481	4	REAL
double	480/481	8	DOUBLE PRECISION
single-character form	452/453	1	CHAR(1)

Table 88. C declarations mapped to typical SQL data types (continued)

C Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
NUL-terminated character form	460/461	length	VARCHAR (length - 1)
VARCHAR structured form	448/449, 456/457	length	VARCHAR (length)
single-graphic form	468/469	1	GRAPHIC(1)
NUL-terminated graphic form (wchar_t)	400/401	length	VARGRAPHIC (length - 1)
VARGRAPHIC structured form	464/465, 472/473	length	VARGRAPHIC (length)
SQLTYPE IS CLOB	408/409	length	CLOB (length)
SQLTYPE IS DBCLOB	412/413	length	DBCLOB (length)
SQLTYPE IS BLOB	404/405	length	BLOB (length)
SQLTYPE IS CLOB_LOCATOR	964/965	4	CLOB locator [187]
SQLTYPE IS DBCLOB_LOCATOR	968/969	4	DBCLOB locator [187]
SQLTYPE IS BLOB_LOCATOR	960/961	4	BLOB locator [187]
SQLTYPE IS XML AS CLOB	408/409	0	XML
SQLTYPE IS XML AS DBCLOB	412/413	0	XML
SQLTYPE IS XML AS BLOB	404/405	0	XML

The following table can be used to determine the C data type that is equivalent to a given SQL data type.

Table 89. SQL data types mapped to typical C declarations

SQL Data Type	C Data Type	Notes
SMALLINT	short int	
INTEGER	long int	
BIGINT	long long int	
DECIMAL(p,s) or NUMERIC(p,s)	decimal	Use double if no exact equivalent.
REAL	float	
DOUBLE PRECISION	double	
DECFLOAT(16)	Not Supported	
DECFLOAT(34)	Not Supported	
CHAR(1)	single-character form	
CHAR(n)	no exact equivalent	If n>1, use NUL-terminated character form

187. Do not use this data type as a column type.

Table 89. SQL data types mapped to typical C declarations (continued)

SQL Data Type	C Data Type	Notes
VARCHAR(n)	NUL-terminated character form	Allow at least n + 1 to accommodate the NUL-terminator. If data can contain character NULs (\0), use VARCHAR structured form. *n* is a positive integer. The maximum value of *n* is 32 672. See Table 60 on page 964 for more information.
	VARCHAR structured form	*n* is a positive integer. The maximum value of *n* is 32 672. See Table 60 on page 964 for more information.
CLOB(n)	SQL TYPE IS CLOB(n)	*n* is a positive integer. The maximum value of *n* is 2 147 483 647. See Table 60 on page 964 for more information.
GRAPHIC(1)	single-graphic form	
GRAPHIC(n)	no exact equivalent	If n>1, use NUL-terminated graphic form
VARGRAPHIC(n)	NUL-terminated graphic form	Allow at least n + 1 to accommodate the NUL-terminator. If data can contain graphic NUL values (\0\0), use VARGRAPHIC structured form. *n* is a positive integer. The maximum value of *n* is 16 336. See Table 60 on page 964 for more information.
	VARGRAPHIC structured from	*n* is a positive integer. The maximum value of *n* is 16 336. See Table 60 on page 964 for more information.
DBCLOB(n)	SQL TYPE IS DBCLOB(n)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 60 on page 964 for more information.
BLOB(n)	SQL TYPE IS BLOB(n)	*n* is a positive integer. The maximum value of *n* is 2 147 483 647. See Table 60 on page 964 for more information.
DATE	NUL-terminated character form	Allow at least 11 characters to accommodate the NUL-terminator.
	VARCHAR structured form	Allow at least 10 characters.
TIME	NUL-terminated character form	Allow at least 7 characters (9 to include seconds) to accommodate the NUL-terminator.
	VARCHAR structured form	Allow at least 6 characters; 8 to include seconds.
TIMESTAMP	NUL-terminated character form	Allow at least 20 characters (27 to include microseconds at full precision) to accommodate the NUL-terminator.
	VARCHAR structured form	Allow at least 19 characters; 26 to include microseconds at full precision.

Table 89. SQL data types mapped to typical C declarations (continued)

SQL Data Type	C Data Type	Notes
XML	SQL TYPE IS CLOB(n)	*n* is a positive integer. The maximum value of *n* is 2 147 483 647. See Table 60 on page 964 for more information.
	SQL TYPE IS DBCLOB(n)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 60 on page 964 for more information.
	SQL TYPE IS BLOB(n)	*n* is a positive integer. The maximum value of *n* is 2 147 483 647. See Table 60 on page 964 for more information.

Chapter 16. Coding SQL statements in COBOL applications

This section describes the programming techniques that are unique to coding SQL statements within a COBOL program.

Defining the SQL communications area in COBOL

A COBOL program that contains SQL statements must include one or both of the following:

- An SQLCODE variable[189] declared as PICTURE S9(9) BINARY, PICTURE S9(9) COMP-4, or PICTURE S9(9) COMP [188]
- An SQLSTATE variable[189] declared as PICTURE X(5)

or,

- An SQLCA (which contains an SQLCODE and SQLSTATE variable).

The SQLCODE and SQLSTATE values are set by the database manager after each SQL statement is executed. An application can check the SQLCODE or SQLSTATE value to determine whether the last SQL statement was successful. See Chapter 13, "SQLSTATE values—common return codes," on page 997 for more information.

The SQLCA can be coded in a COBOL program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

The SQLCA must not be defined within an SQL declare section. See Chapter 11, "SQLCA (SQL communication area)," on page 981 and "INCLUDE" on page 808 for more information.

The SQLSTATE, SQLCODE, and SQLCA variables must appear in the WORKING-STORAGE SECTION or LINKAGE SECTION of the program and can be placed wherever a record description entry can be specified in those sections.

Defining SQL descriptor areas in COBOL

The following statements require an SQLDA:

EXECUTE...USING DESCRIPTOR *descriptor-name*
FETCH...USING DESCRIPTOR *descriptor-name*
OPEN...USING DESCRIPTOR *descriptor-name*
DESCRIBE *statement-name* INTO *descriptor-name*
PREPARE *statement-name* INTO *descriptor-name*
CALL...USING DESCRIPTOR *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name.

188. In DB2 for LUW, the SQLCODE variable must be declared as COMP-5.

189. In DB2 for z/OS, the STDSQL(YES) option must be in effect to declare the SQLSTATE and SQLCODE variables. In DB2 for LUW, the LANGLEVEL SQL92E option must be used to declare the SQLSTATE and SQLCODE variables.

The SQLDA can be coded in a COBOL program either directly or by using the SQL INCLUDE statement. The SQLDA must not be defined within an SQL declare section. See Chapter 12, "SQLDA (SQL descriptor area)," on page 985 and "INCLUDE" on page 808 for more information. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLDA END-EXEC.
```

SQLDA declarations must appear in the WORKING-STORAGE SECTION or LINKAGE SECTION of the program and can be placed wherever a record description entry can be specified in those sections.

Embedding SQL statements in COBOL

SQL statements can be coded in COBOL program sections as follows:

SQL Statement	Program Section
BEGIN DECLARE SECTION END DECLARE SECTION	WORKING-STORAGE SECTION or LINKAGE SECTION
INCLUDE SQLCA INCLUDE SQLDA	WORKING-STORAGE SECTION
DECLARE CURSOR INCLUDE name	DATA DIVISION or PROCEDURE DIVISION
Other	PROCEDURE DIVISION

SQL statements must not be coded in COBOL programs with more than one PROCEDURE DIVISION.

Each SQL statement in a COBOL program must begin with EXEC SQL and end with END-EXEC. If the SQL statement appears between two COBOL statements, the period is optional and might not be appropriate. The EXEC SQL keywords must appear all on one line, but the remainder of the statement can appear on the next and subsequent lines.

For example, an UPDATE statement coded in a COBOL program might be coded as follows:

```
EXEC SQL
   UPDATE DEPARTMENT
   SET MGRNO = :MGR-NUM
   WHERE DEPTNO = :INT-DEPT
END-EXEC.
```

Comments

In addition to SQL comments (--), COBOL comment lines (* in column 7) can be included within embedded SQL statements, except between the keywords EXEC and SQL. [190]

190. In DB2 for z/OS, the STDSQL(YES) option must be in effect to use SQL comments.

Continuation for SQL statements

The line continuation rules for SQL statements are the same as those for other COBOL statements, except that EXEC SQL must be specified within one line.

If a string constant is continued from one line to the next, the first nonblank character in the next line must be either an apostrophe or a quotation mark. In DB2 for LUW this character must be an apostrophe. Identifiers that are not delimited cannot be continued. If a delimited identifier is continued from one line to the next, the first nonblank character in the next line must be either an apostrophe or a quotation mark. In DB2 for LUW this character must be a quotation mark.

Cursors

The DECLARE CURSOR statement must precede all statements that explicitly refer to the cursor by name.

Including code

SQL statements or COBOL host variable declaration statements can be included by embedding the following SQL statement at the point in the source code where the statements are to be embedded:

```
EXEC SQL INCLUDE name END-EXEC.
```

COBOL COPY statements cannot be used to include SQL statements or declarations of COBOL variables that are referenced in SQL statements.

Margins

SQL statements must be coded in columns 12 through 72.

Names

Any valid COBOL variable name can be used for a host variable, as long as it:

- does not contain DBCS characters
- does not begin with 'DB2', 'DSN', 'RDI', or 'SQL' in any combination of uppercase or lowercase letters (these names are reserved for the database manager).

It is recommended that FILLER not be used as a variable name. Name all fields within a COBOL structure to avoid unexpected results from using structures that contain FILLER.

Access plan names must not start with 'DSN'. External entry names must not start with 'DSN', 'RDI', or 'SQL'.

For information on the length of a host identifier, see Table 58 on page 960.

Statement labels

Executable SQL statements in the PROCEDURE DIVISION can be preceded with a paragraph name.

Handling SQL errors and warnings in COBOL

The SQL WHENEVER statement tests the result of every SQL statement within its scope for an error or warning condition. The target for the GOTO clause in an SQL WHENEVER statement must be a section name or unqualified paragraph name in the PROCEDURE DIVISION.

The stand-alone SQLSTATE and SQLCODE or information in the SQLCA can also be used in the detection or further handling of error and warning conditions. See Chapter 11, "SQLCA (SQL communication area)," on page 981 for more information.

Using host variables in COBOL

A host variable used in an SQL statement must be explicitly declared prior to the first use of the host variable in an SQL statement.

The COBOL statements that are used to define the host variables must be preceded by a BEGIN DECLARE SECTION statement and followed by an END DECLARE SECTION statement.

All host variables within an SQL statement must be preceded by a colon (:).

The names of host variables must be unique within the program, even if the host variables are in different blocks or procedures.

Host variables must not be arrays or array elements unless they are used to represent indicator arrays or indicator variables. Host variables must not be records or elements.

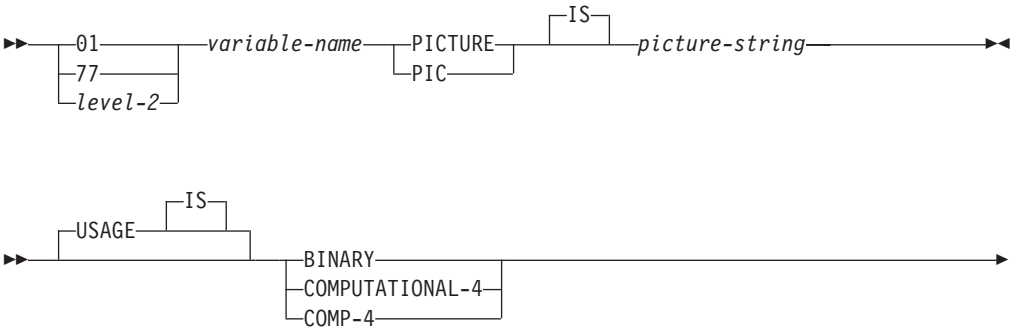
Declaring host variables in COBOL

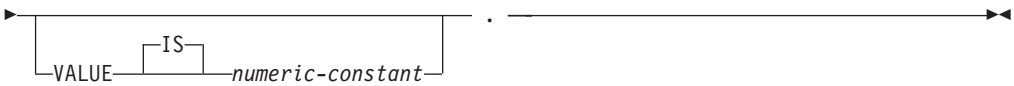
Only a subset of valid COBOL declarations are recognized as valid host variable declarations.

Numeric host variables

The following figures show the syntax for valid integer host variable declarations.

BIGINT, INTEGER, and SMALLINT





Notes:

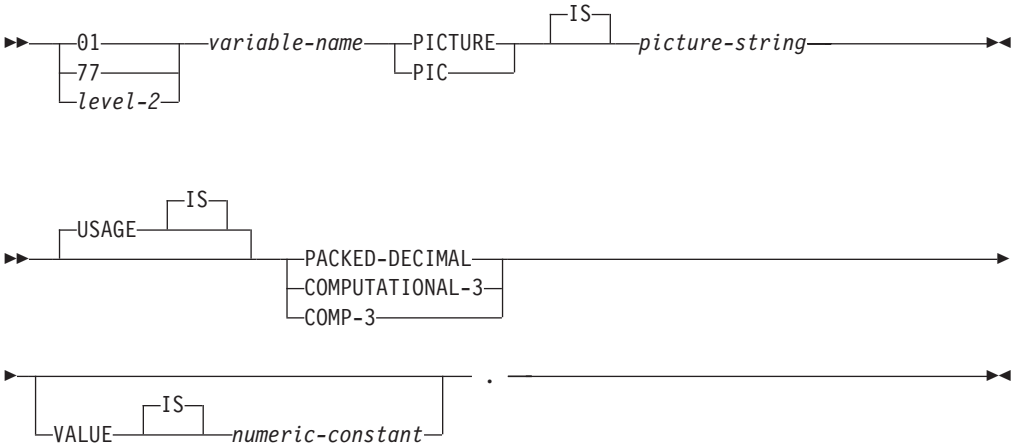
- BINARY, COMPUTATIONAL-4, COMP-4, are equivalent. COMPUTATIONAL-4 and COMP-4 are IBM extensions that are not supported in ISO/ANS COBOL. The *picture-string* associated with these types must be either S9(4), S9999, S9(9), S999999999. S9(18) or S999999999999999999.

  G In DB2 for LUW, these declarations are not supported; COMPUTATIONAL-5 or
  G COMP-5 must be used instead.

- *level-2* indicates a COBOL level between 2 and 48.

The following figure shows the syntax for valid decimal host variable declarations.

DECIMAL

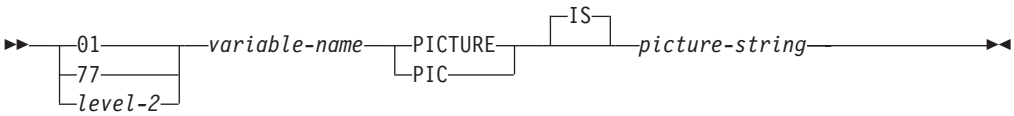


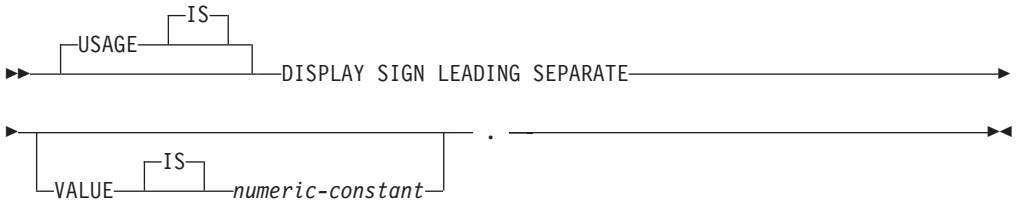
Notes:

- PACKED-DECIMAL, COMPUTATIONAL-3, and COMP-3 are equivalent. COMPUTATIONAL-3 and COMP-3 are IBM extensions that are not supported in ISO/ANS COBOL. The *picture-string* associated with these types must have the form S9(i)V9(d) (or S9...9V9...9, with *i* and *d* instances of 9). ISO/ANSI COBOL restricts i + d to be less than or equal to 18.
- *level-2* indicates a COBOL level between 2 and 48.

The following figure shows the syntax for valid numeric host variable declarations.

NUMERIC



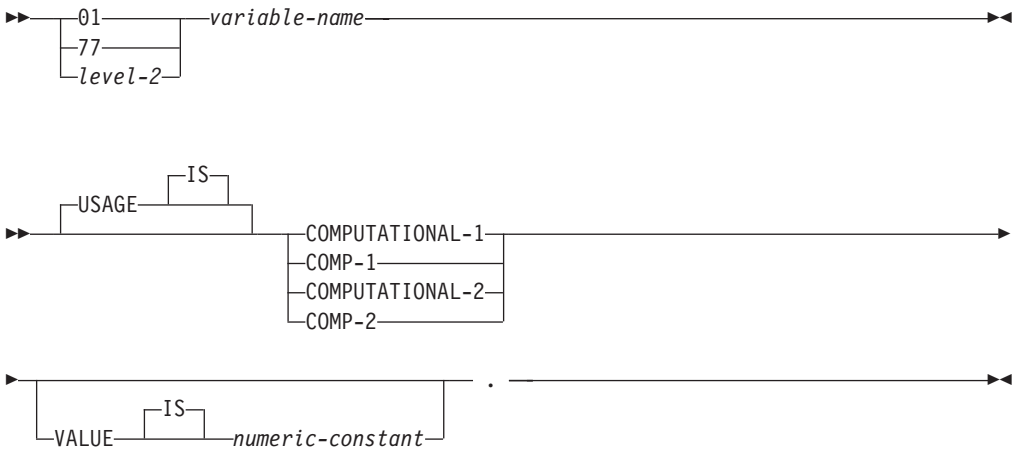


Notes:

- The picture-string associated with SIGN LEADING SEPARATE must have the form S9(i)V9(d) (or S9...9V9...9, with *i* and *d* instances of 9). ISO/ANSI COBOL G restricts i + d to be less than or equal to 18. In DB2 for LUW, SIGN LEADING G SEPARATE is not supported.
- *level-2* indicates a COBOL level between 2 and 48.

The following figure shows the syntax for valid floating-point host variable declarations.

Floating point

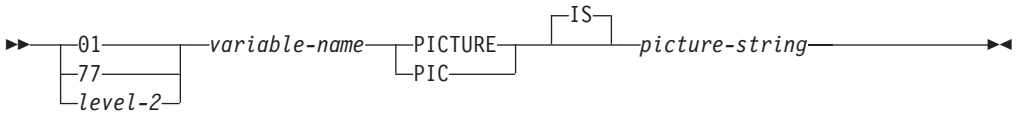


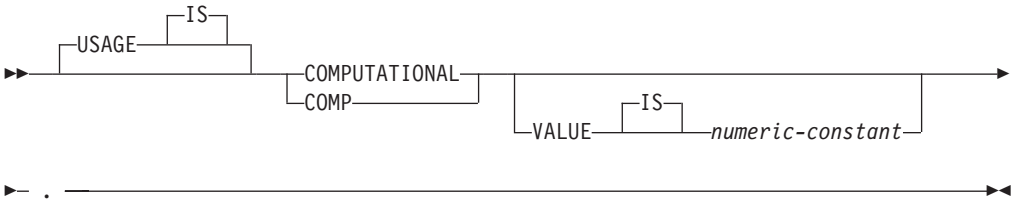
Notes:

- COMPUTATIONAL-1 and COMP-1 are equivalent. COMPUTATIONAL-2 and COMP-2 are equivalent.
- *level-2* indicates a COBOL level between 2 and 48.

The following figure shows the syntax for other valid numeric host variable declarations.

Other numeric





Notes:

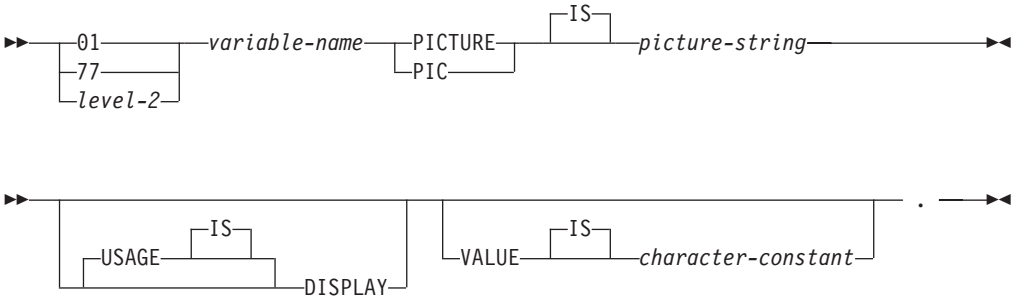
- COMPUTATIONAL and COMP are equivalent. The picture strings associated with these and the data types they represent are product-specific. Therefore, do not use COMP and COMPUTATIONAL in a portable application.
- *level-2* indicates a COBOL level between 2 and 48.

Character host variables (excluding CLOB)

There are two valid non-LOB forms of character host variables:

- Fixed-Length Strings
- Varying-Length Strings

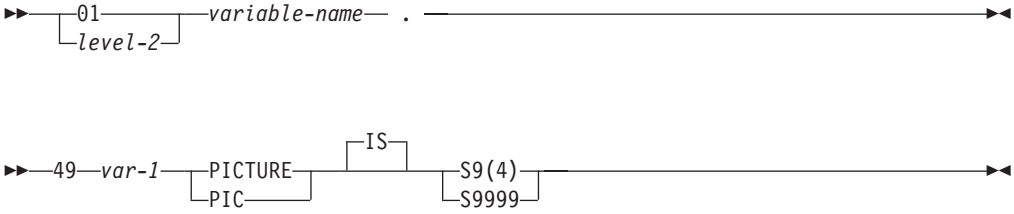
Fixed-length character strings

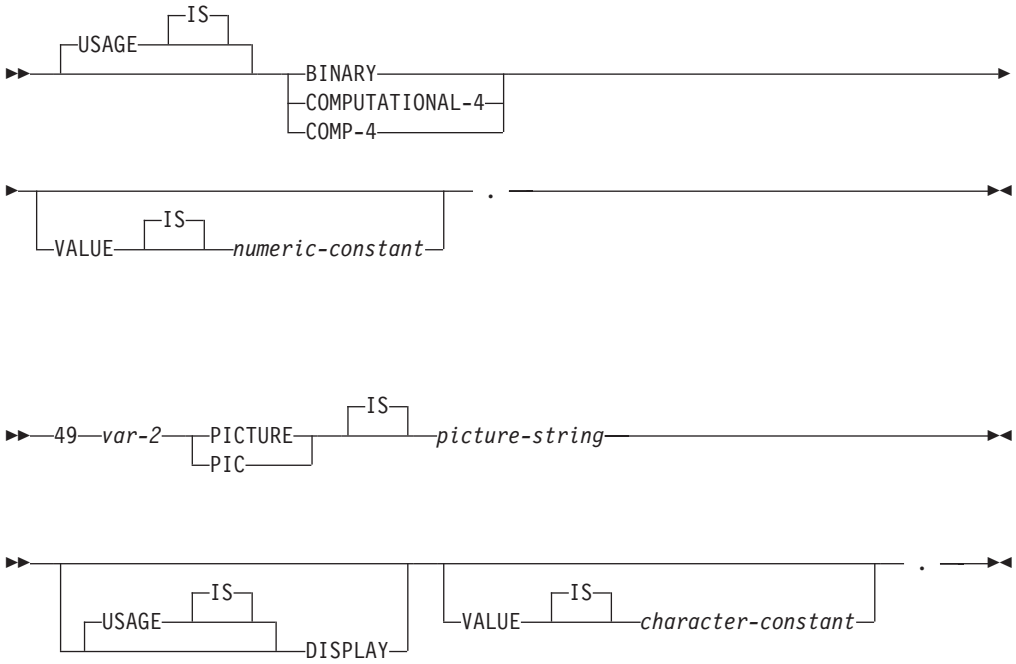


Notes:

- The *picture-string* associated with this form must be X(*m*) (or XXX...X, with *m* instances of X). *m* must be no greater than the maximum length of CHAR. See Table 60 on page 964 for more information.
- *level-2* indicates a COBOL level between 2 and 48.

Varying-length character strings





Notes:

- The *picture-string* associated with this form must be X(m) (or XXX...X, with *m* instances of X). *m* can be no greater than the maximum length of VARCHAR. See Table 60 on page 964 for more information.

G In DB2 for LUW, COMP-5 must be used in place of COMP-4.

  Note that the database manager will use the full size of the S9(4) variable even though ISO/ANSI COBOL only recognizes values up to 9999. This can cause data truncation errors when COBOL statements are being executed and may effectively limit the maximum length of variable-length character strings to 9999.

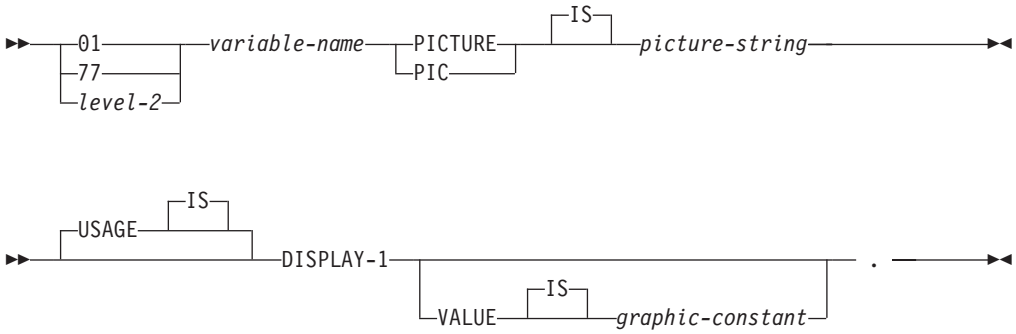
- *var-1* and *var-2* cannot be used as host variables.
- *level-2* indicates a COBOL level between 2 and 48.

Graphic host variables (excluding DBCLOB)

There are two valid non-LOB forms for graphic host variables:

- Fixed-Length Strings
- Varying-Length Strings

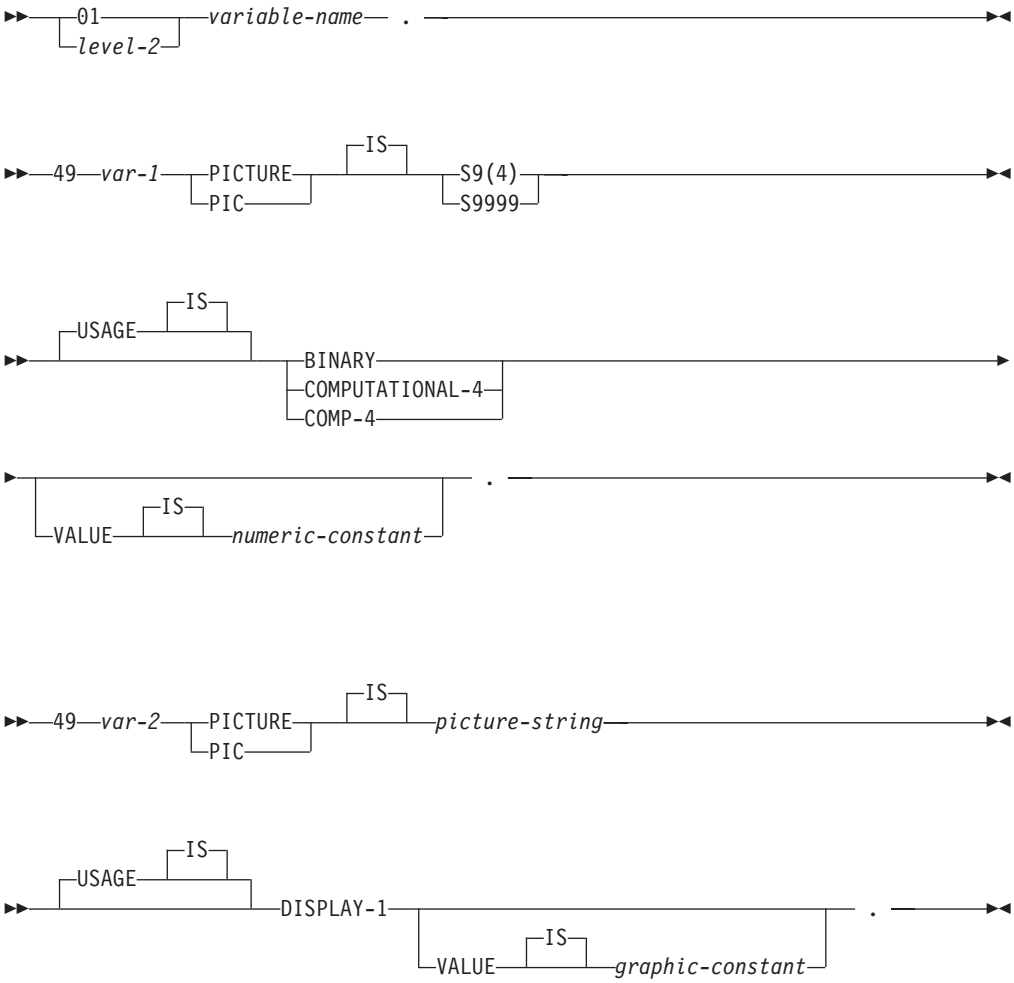
Fixed-length graphic strings



Notes:

- The *picture-string* associated with this form must be G(*m*) (or GGG...G, with *m* instances of G. *m* must be no greater than the maximum length of GRAPHIC. See Table 60 on page 964 for more information.
- *level-2* indicates a COBOL level between 2 and 48.

Varying-length graphic strings



Notes:

- The *picture-string* associated with this form must be G(*m*) (or GGG...G, with *m* instances of G). *m* must be no greater than the maximum length of VARGRAPHIC. See Table 60 on page 964 for more information.

  Note that the database manager will use the full size of the S9(4) variable even though ISO/ANSI COBOL only recognizes values up to 9999. This can cause data truncation errors when COBOL statements are being executed and may effectively limit the maximum length of variable-length graphic strings to 9999.

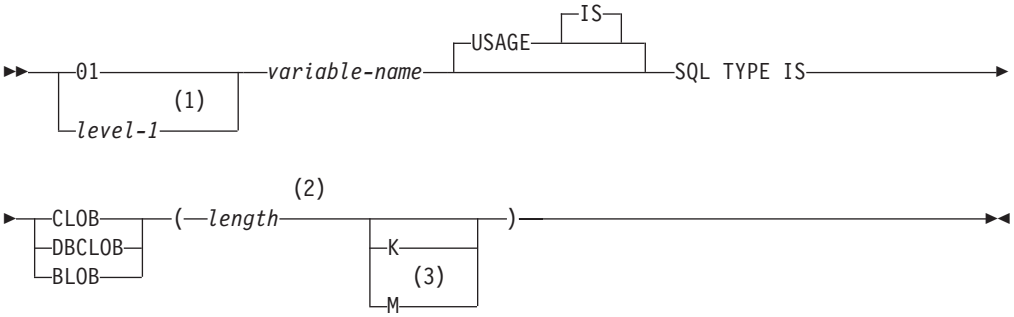
G In DB2 for LUW, COMP-5 must be used in place of COMP-4.

- *var*-1 and *var*-2 cannot be used as host variables.
- *level-2* indicates a COBOL level between 2 and 48.

LOB host variables

COBOL does not have variables that correspond to the SQL data types for LOBs (large objects). To define host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source.

LOB host variable



Notes:

1 *level-1* indicates a COBOL level between 2 and 48.

2 *length* must be an integer constant that is greater than 0 and no greater than the maximum length of CLOB. See Table 60 on page 964 for more information. The maximum value for *length* is further restricted if K or M or if DBCLOB is specified.

3 K multiplies *length* by 1024. M multiplies *length* by 1 048 576.

Examples: *Example 1:* The following declaration:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

Results in the generation of the following structure:

```
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-CLOB-DATA PIC X(131072000).
```

Example 2: The following declaration:

```
01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

Results in the generation of the following structure:

```
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

Example 3: The following declaration:

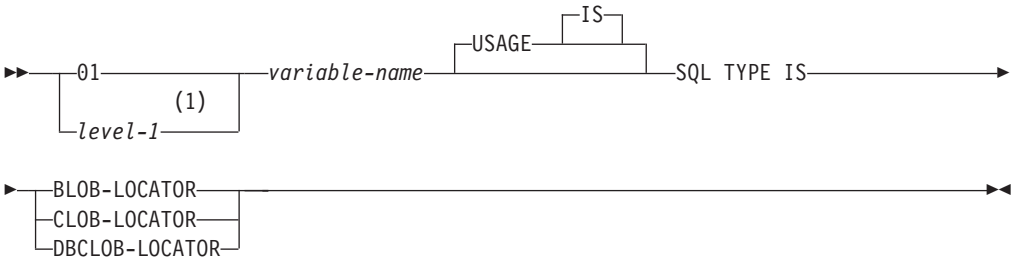
```
 01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

Results in the generation of the following structure:

```
 01 MY-BLOB.
    49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
    49 MY-BLOB-DATA PIC X(2097152).
```

LOB locators

LOB locator



Notes:

1 *level-1* indicates a COBOL level between 2 and 48.

Example: The following declaration (other LOB locator types are similar):

```
01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

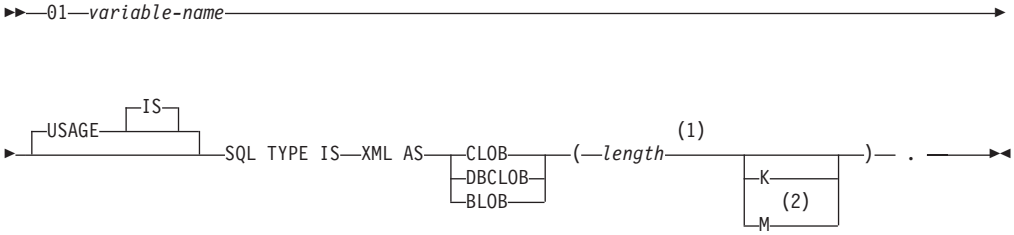
Results in the generation of the following declaration:

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

XML host variables

COBOL does not have variables that correspond to the SQL data type for XML. To create host variables that can be used with this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source member.

XML host variables



Notes:

1 *length* must be an integer constant that is greater than 0 and no greater than 15,728,640 for a CLOB and 7,864,320 for a DBCLOB. The maximum value for *length* is further restricted if K or M

2 K multiplies *length* by 1024. M multiplies *length* by 1 048 576.

Examples

The following declaration:

```
01 MY-XML SQL TYPE IS XML AS CLOB(5000).
```

Results in the generation of the following structure:

```
01 MY-XML.
   49 MY-XML-LENGTH PIC 9(9) BINARY.
   49 MY-XML-DATA PIC X(5000).
```

Indicator variables in COBOL

An indicator variable is a two-byte integer (PIC S9(4) USAGE BINARY). On retrieval, an indicator variable is used to show whether its associated host variable has been assigned a null value. On assignment to a column, a negative indicator variable is used to assign a null value.

See "References to host variables" on page 116 for more information on the use of indicator variables.

Indicator variables are declared in the same way as host variables, and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

Example: Given the statement:

```
EXEC SQL FETCH CLS_CURSOR INTO
                                :DAY-VAR :DAY-IND,
                                :BGN-VAR :BGN-IND,
                                :END-VAR :END-IND
END-EXEC.
```

Variables can be declared as follows:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
77 CLS-CD    PIC X(7).
77 DAY-VAR   PIC S9(4) BINARY.
77 BGN-VAR   PIC X(8).
77 END-VAR   PIC X(8).
77 DAY-IND   PIC S9(4) BINARY.
77 BGN-IND   PIC S9(4) BINARY.
77 END-IND   PIC S9(4) BINARY.
EXEC SQL END DECLARE SECTION END-EXEC.
```

Declaring host structures in COBOL

A COBOL host structure is a named set of host variables that is defined in the program's WORKING-STORAGE SECTION or LINKAGE SECTION. COBOL host structures have a maximum of two levels, even though the host structure might occur within a multilevel structure. One exception is the declaration of a varying-length character string, which must be level 49.

A host structure name can be a group name whose subordinate levels name elementary data items. In the following example, B is the name of a host structure consisting of the elementary items C1 and C2.

```
01 A
  02 B
    03 C1 PICTURE ...
    03 C2 PICTURE ...
```

When writing an SQL statement using a qualified host variable name (for example, to identify a field within a structure), use the name of the structure followed by a period and the name of the field. For example, specify B.C1 rather than C1 OF B or C1 IN B.

A host structure is considered complete if any of the following items are found:

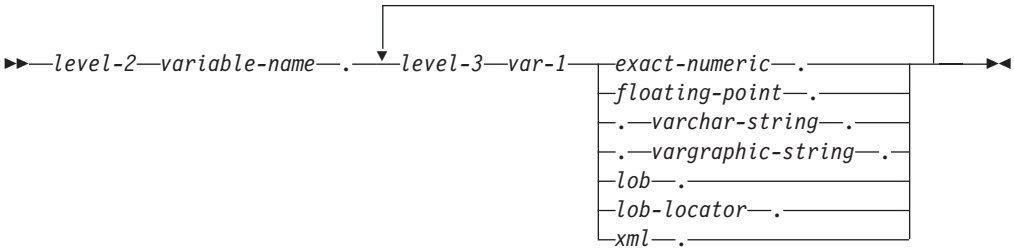
- A COBOL item that must begin in area A
- Any SQL statement (except SQL INCLUDE).

- Any SQL statement within an included member.

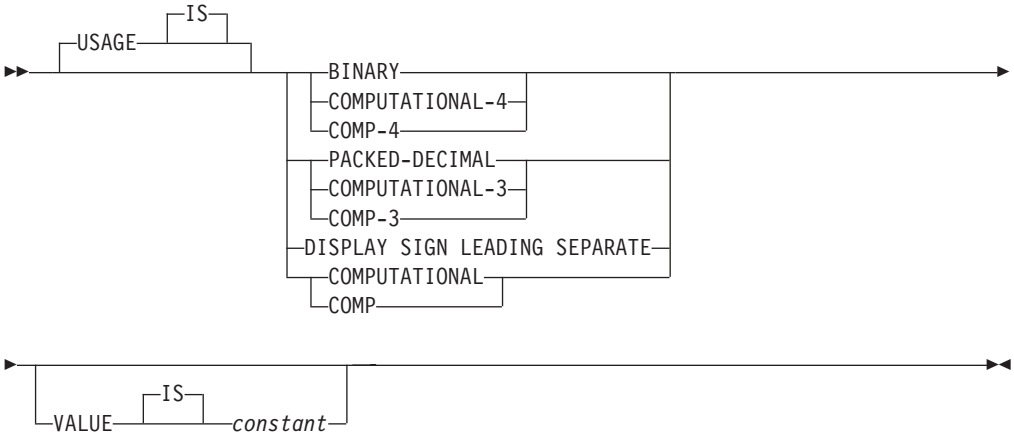
Name all fields within a COBOL structure to avoid unexpected results that might result from using structures that contain FILLER.

The following figure shows the syntax for valid host structures.

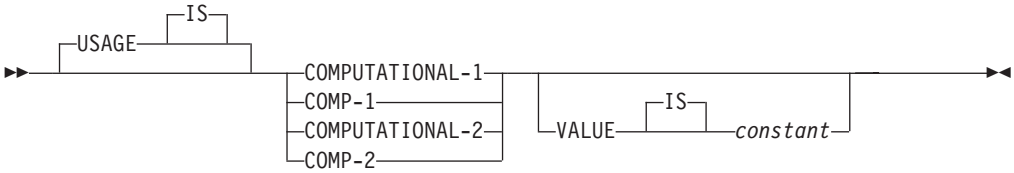
Host structures



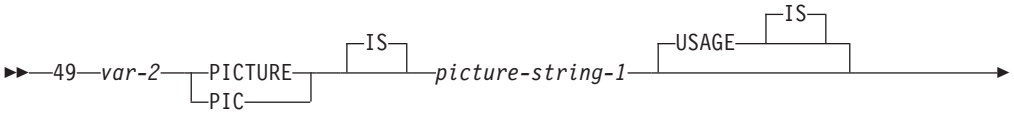
exact-numeric:

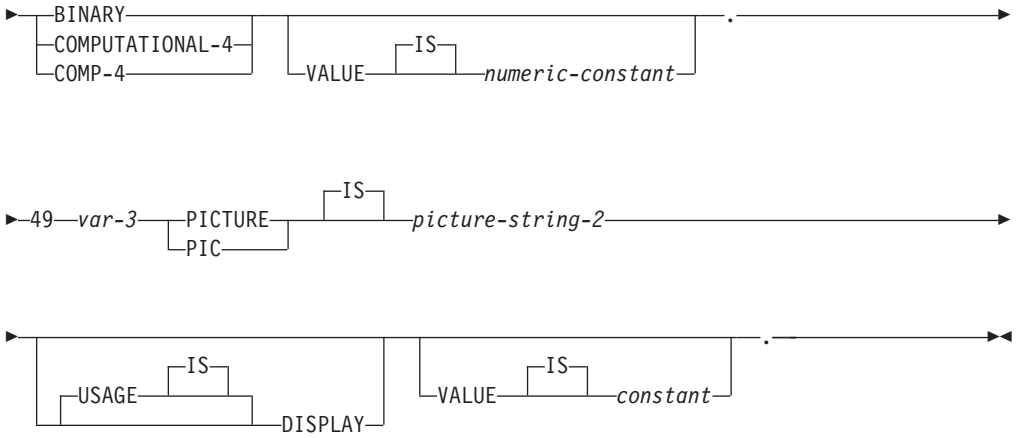


floating-point:

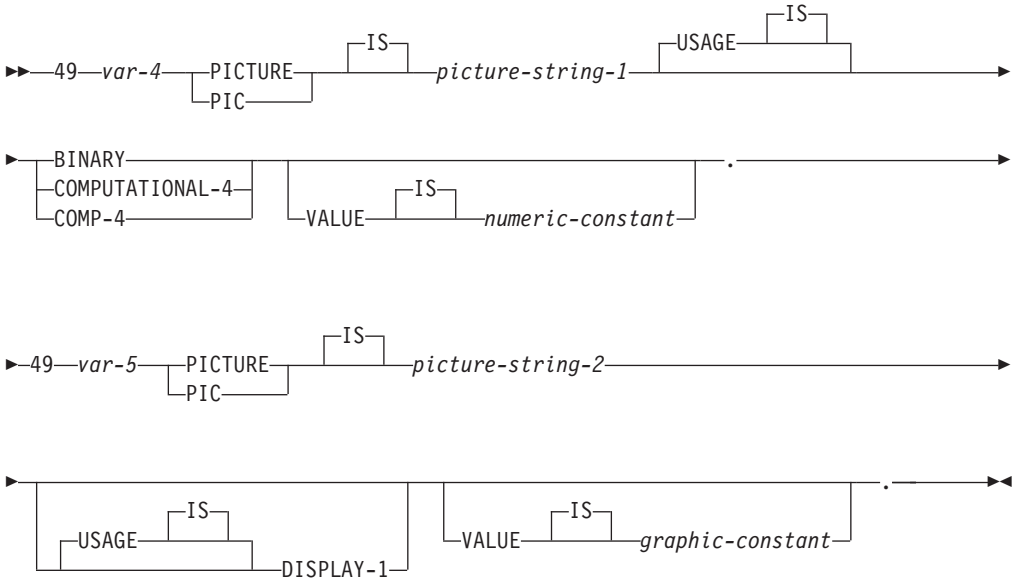


varchar-string

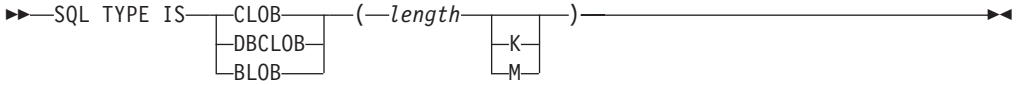




vargraphic-string

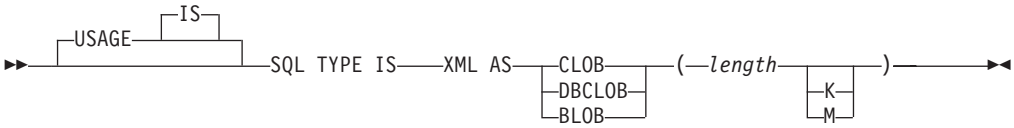


lob



lob-locator

xml



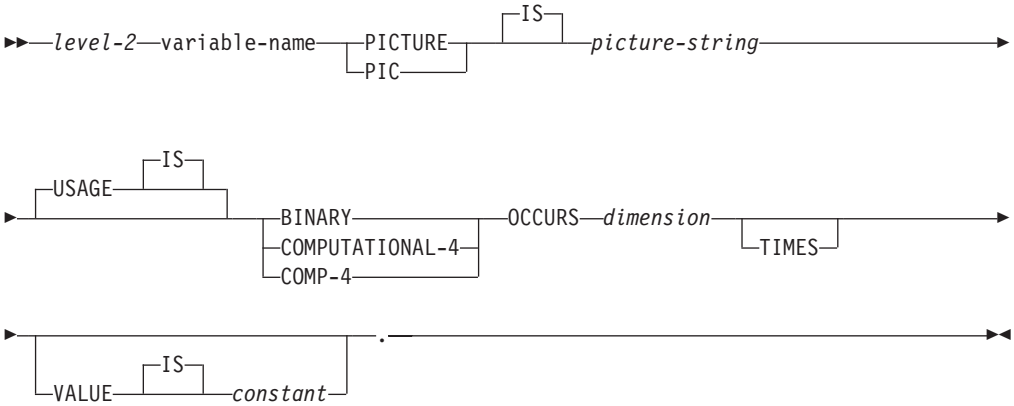
Notes:

- *level-2* indicates a COBOL level between 1 and 47.
- *level-3* indicates a COBOL level between 2 and 48.
- G In DB2 for LUW, COMP-5 must be used in place of COMP-4.

Host structure indicator array

The following figure shows the syntax for valid indicator array declarations.

Host structure indicator array



Note:

1. Dimension must be an integer between 1 and 32767.
2. *level-2* must be an integer between 2 and 48.
3. BINARY, COMPUTATIONAL-4, and COMP-4 are equivalent. COMPUTATIONAL-4 and COMP-4 are IBM extensions that are not supported in ISO/ANSI COBOL. The *picture-string* associated with these types must have the form S9(i) (or S9...9, with i instances of 9). i must be less than or equal to 4. G In DB2 for LUW, COMP-5 must be used in place of COMP-4.

Determining equivalent SQL and COBOL data types

The base SQLTYPE and SQLLEN of host variables are determined according to the following table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.

Table 90. COBOL declarations mapped to typical SQL data types

COBOL Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
COMP-1	480/ 481	4	REAL

Table 90. COBOL declarations mapped to typical SQL data types (continued)

COBOL Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
COMP-2	480/ 481	8	DOUBLE PRECISION
S9(*i*)V9(*d*) COMP-3 or S9(*i*)V9(*d*) PACKED-DECIMAL	484/ 485	*i*+*d* in byte 1, *d* in byte 2	DECIMAL(*i*+*d*,*d*)
S9(*i*)V9(*d*) DISPLAY SIGN LEADING SEPARATE [192]	504/ 505	*i*+*d* in byte 1, *d* in byte 2	No exact equivalent. Use DECIMAL(*i*+*d*,*d*) or NUMERIC (*i*+*d*,*d*)
S9(4) COMP-4 [191] or S9(4) BINARY	500/ 501	2	SMALLINT
S9(9) COMP-4 [191] or S9(9) BINARY	496/ 497	4	INTEGER
S9(18) COMP-4 [191] or S9(18) BINARY	492/ 493	8	BIGINT
Fixed-length character data	452/ 453	length	CHAR(length)
Varying-length character data	448/ 449, 456/ 457	length	VARCHAR(length)
Fixed-length graphic data	468/ 469	length	GRAPHIC(length)
Varying-length graphic data	464/ 465, 472/ 473	length	VARGRAPHIC(length)
USAGE IS SQL TYPE IS CLOB(*n*) *n* < 2147483648	408/ 409	length	CLOB(length)
USAGE IS SQL TYPE IS DBCLOB(*n*) *n* < 1073741824	412/ 413	length	DBCLOB(length)
USAGE IS SQL TYPE IS BLOB(*n*) *n* < 2147483648	404/ 405	length	BLOB(length)
SQL TYPE IS CLOB-LOCATOR	964/ 965	4	CLOB locator [193]
SQL TYPE IS DBCLOB-LOCATOR	968/ 969	4	DBCLOB locator [193]
SQL TYPE IS BLOB-LOCATOR	960/ 961	4	BLOB locator [193]
USAGE IS SQL TYPE IS XML AS CLOB(*n*) *n* < 2147483648	404/ 405	0	XML
USAGE IS SQL TYPE IS XML AS DBCLOB(*n*) *n* < 1073741824	408/ 409	0	XML
USAGE IS SQL TYPE IS XML AS BLOB(*n*) *n* < 2147483648	404/ 405	0	XML

The following table can be used to determine the COBOL data type that is equivalent to a given SQL data type:

191. In DB2 for LUW, COMP-5 must be used instead of COMP-4.

192. In DB2 for LUW, DISPLAY SIGN LEADING SEPARATE is not supported.

193. Do not use this data type as a column type.

Table 91. SQL data types mapped to typical COBOL declarations

SQL Data Type	COBOL Data Type	Notes
SMALLINT	S9(4) COMP-4	
INTEGER	S9(9) COMP-4	
BIGINT	S9(18) COMP-4	
DECIMAL(*p*,*s*) or NUMERIC(p,s)	If *p* < 19: S9(*p*-*s*)V9(*s*) PACKED-DECIMAL or S9(*p*-*s*)V9(*s*) DISPLAY SIGN LEADING SEPARATE If *p* > 18: no exact equivalent	0<=*s*<=*p*<=18, where *s* is the scale and *p* is the precision. If *s*=0, use S9(*p*) or S9(*p*)V. If *s*=*p*, use SV9(*s*). Use COMP-2
REAL	COMP-1	
DOUBLE PRECISION	COMP-2	
DECFLOAT(16)	Not supported	
DECFLOAT(34)	Not supported	
CHAR(*n*)	fixed-length character string	*n* is a positive integer. The maximum value of *n* is 254. See Table 60 on page 964 for more information.
VARCHAR(*n*)	varying-length character string	*n* is a positive integer. The maximum value of *n* is 32 672. See Table 60 on page 964 for more information.
CLOB(*n*)	USAGE IS SQL TYPE IS CLOB(*n*)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 60 on page 964 for more information.
GRAPHIC(*n*)	fixed-length graphic string	*n* is a positive integer. The maximum value of *n* is 127. See Table 60 on page 964 for more information.
VARGRAPHIC(*n*)	varying-length graphic string	*n* is a positive integer. The maximum value of *n* is 16 336. See Table 60 on page 964 for more information.
DBCLOB(*n*)	USAGE IS SQL TYPE IS DBCLOB(*n*)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 60 on page 964 for more information.
BLOB(*n*)	USAGE IS SQL TYPE IS BLOB(*n*)	*n* is a positive integer. The maximum value of *n* is 2 147 483 647.
DATE	fixed-length character string	Allow at least 10 characters.
TIME	fixed-length character string	Allow at least 6 characters; 8 to include seconds.

Table 91. SQL data types mapped to typical COBOL declarations (continued)

SQL Data Type	COBOL Data Type	Notes
TIMESTAMP	fixed-length character string	Allow at least 19 characters; 26 to include microseconds at full precision.
XML	SQL TYPE IS XML AS CLOB(n)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 60 on page 964 for more information.
	SQL TYPE IS XML AS DBCLOB(n)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 60 on page 964 for more information.
	SQL TYPE IS XML AS BLOB(n)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 60 on page 964 for more information.

Notes on COBOL variable declaration and usage

Any level 77 data description entry can be followed by one or more REDEFINES entries. However, the names in these entries cannot be used in SQL statements.

The COBOL declarations for SMALLINT, INTEGER, and BIGINT data types are expressed as a number of decimal digits. The database manager uses the full size of the integers and can place larger values in the host variable than would be allowed in the specified number of digits in the COBOL declaration. However, this can cause data truncation or size errors when COBOL statements are being executed. The size of numbers in the application must be within the declared number of digits.

Chapter 17. Coding SQL statements in Java applications

Support for embedded static SQL in Java applications is commonly referred to as "SQLJ". This appendix also makes use of that term.

Defining the SQL communications area in Java

A Java program containing SQL statements does not use any Java class corresponding to an SQLCA to inform an application of errors and warnings resulting from the execution of its contained SQL statements. Instead, Java programs are made aware of errors and warnings as described in “Handling SQL errors and warnings in Java” on page 1111.

Defining SQL descriptor areas in Java

A Java program containing SQL statements does not use any Java class corresponding to an SQLDA to associate the application's variables with the input and output parameters of its contained SQL statements. Due to the absence of an SQLDA, none of the SQL statements that include a USING DESCRIPTOR clause are able to specify that clause. Instead, Java programs directly embed host variables and expressions in the SQL statements as described in “Using host variables and expressions in Java” on page 1104

Embedding SQL statements in Java

In a Java program, static SQL statements used for database access are contained in *SQLJ clauses*. SQLJ clauses containing SQL statements are called *executable clauses*. SQLJ clauses that result in declarations of Java classes needed by the executable clauses are called *declaration clauses*, and the classes that result are called *generated classes*.

An executable clause may appear anywhere in a program that a Java statement is permitted. An executable clause begins with the characters #sql, terminates with a semicolon (;), and contains an SQL statement enclosed in braces, {}. The SQL statement itself has no terminating character. An example executable clause is:

```
#sql {DELETE FROM EMPLOYEE};
```

A declaration clause may appear anywhere in a program that a Java class declaration is permitted. A declaration clause begins with the characters #sql, terminates with a semicolon (;), and contains information used in the generation of either an SQLJ database connection context class or an SQLJ iterator class. An example declaration clause is:

```
#sql public iterator DeptSummary (String, String, BigDecimal);
```

This clause results in the generation of a declaration of a public SQLJ iterator class named DeptSummary that fulfills part of the role a cursor declaration does in other application languages. In this example, an associated cursor would be one involving two character strings and a decimal value in that order.

Before any embedded SQL statements can be executed in an application program, code must be included to accomplish these tasks:

- Import the Java packages for SQLJ run-time support and the JDBC interfaces used by SQLJ.[194]
- Load a JDBC driver.
- Connect to a data source by creating a connection context.
- Optionally, create an execution context.

To import the Java packages for SQLJ and JDBC, these lines are included in the application program:

```
import sqlj.runtime.*;         // SQLJ runtime support
import java.sql.*;             // JDBC interfaces
```

To load the IBM DB2 Driver for JDBC and SQLJ, supported by DB2 for z/OS, DB2 for LUW, and DB2 for i, and register it with the `java.sql.DriverManager`, invoke `Class.forName` with a `java.lang.String` argument specifying "com.ibm.db2.jcc.DB2Driver".

For example:

```
try
{
  Class.forName("com.ibm.db2.jcc.DB2Driver");
}
catch (ClassNotFoundException e)
{
  e.printStackTrace();
}
```

Alternatively, there exist some platform-specific JDBC drivers.

When using any JDBC driver, the necessary contents of the CLASSPATH environmental variable will vary based on the platform and on the driver being used. For further information, see applicable product documentation.

A *connection context* specifies the data source each executable clause is to be executed against. This allows an application to direct individual SQL statements to distinct data sources. Connections contexts are described further in "Connecting to, and using a data source" on page 1101.

An *execution context* provides access to an executable clause's warning information and in the case of a CALL statement to a procedure's returned result sets. It also allows some attributes of a statement's execution to be controlled, such the maximum number or rows returned. The support provided for an execution context to control a statement's execution is platform specific. Further details regarding an execution context's use in returning warning information is provided in "Handling SQL errors and warnings in Java" on page 1111.

In executable clauses, either or both connection contexts and execution contexts are explicitly specified by enclosing them in square brackets, [ ], following the #sql at the beginning of the embedded SQL statement. If both are specified, the connection context is listed first, followed by a comma, followed by the execution context.

Comments

To include comments in an SQLJ program, use either Java comments or SQL comments.

194. SQLJ was designed to coexist with (and in many respects depend on) JDBC. A single application could create a JDBC connection and use that connection to execute dynamic SQL statements through JDBC and embedded static SQL through SQLJ.

- Java comments are denoted by /* */ or //. Java comments can be used outside of SQLJ clauses, wherever the Java language permits them. Within an executable clause, Java comments can only be used in embedded host expressions.
- SQL comments (--) can be used in executable clauses, anywhere except in embedded host expressions.

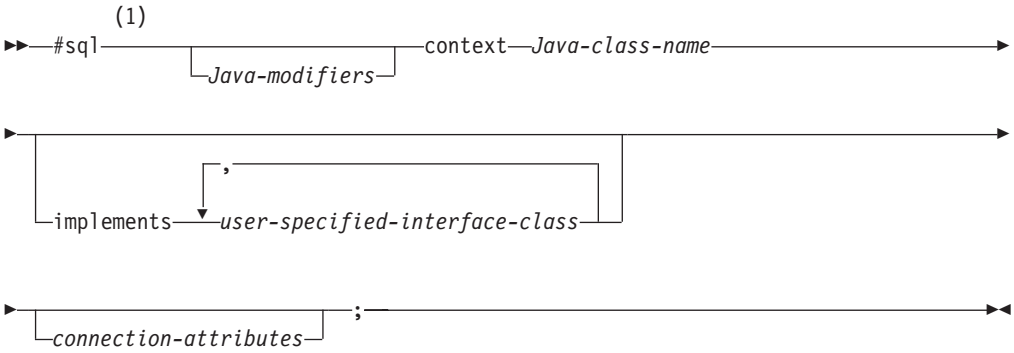
Connecting to, and using a data source

In an SQLJ application, a connection to a data source must be established *before* SQL statements can be executed. A connection to a data source is referred to as a connection context, each of which is an instance of a generated *connection context class* and is declared with a *connection declaration clause*.

Declaring a connection context

A connection declaration clause may appear anywhere in a program that a Java class declaration is permitted.

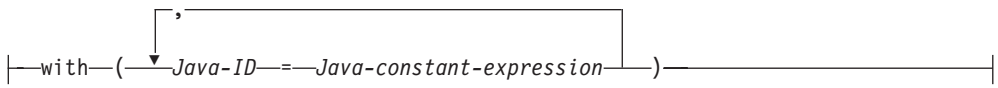
Syntax



Notes:

1 The Java programming language is case-sensitive and lower case is typically used for keywords. For that reason, unlike other SQL keywords, the keywords appearing in a connection context declaration clause are shown in lower case and must appear in the statement in lower case.

connection-attributes:



Description

Java-modifiers
Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
Names the generated connection context. *Java-class-name* must be a valid Java identifier.

implements
The clause specifies one or more user-defined Java interfaces that this connection context implements. Each contained *user-specified-interface-class* must identify a valid Java interface according to Java's rules for use of interfaces.

with
Introduces a set of static attributes of the generated connection context class and the initial value of each such static attribute.

Java-ID
Names a user-defined static attribute of a generated connection context class. *Java-ID* must be a valid Java identifier. The value of *Java-constant-expression*, which supplies that attribute's initial value, is also user-defined.

Initiating and using a connection

After a connection declaration clause has resulted in the generation of a connection context class and the appropriate JDBC driver has been registered with the DriverManager, to initiate a connection to a data source one of the following methods is used:

- Connection method 1:
  1. Invoke the constructor for the connection context class with the following arguments:[195]
     - a `java.lang.String` that specifies the location name associated with the data source,
     - a boolean that specifies whether autoCommit is on or off for the connection.

     In the case of the `java.lang.String` parameter, the general format of a database's location name (also known as its Universal Resource Locator (URL), or Uniform Resource Identifier (URI)) is `jdbc:`*`subprotocol`*`:`*`subname`*. Where subprotocol identifies a JDBC driver, and the format of subname is specific to that driver. However, in cases where a subname specifies the network address of the data source, it will generally have the format `//hostname:port/subsubname`. For further information on the value to specify for this parameter, see applicable product documentation.

  For example, with DB2 for z/OS, to use the first method to set up connection context myConn to access data associated with location NEWYORK and to set autoCommit off, the following steps are taken. First, specify a connection declaration clause to generate a connection context class:

  ```
  #sql context Ctx;
  ```

  Then register a JDBC driver and invoke the constructor for generated class Ctx with arguments jdbc:db2:NEWYORK and `false`:

  ```
  Class.forName("com.ibm.db2.jcc.DB2Driver");
  Ctx myConn=new Ctx("jdbc:db2:NEWYORK",false);
  ```

  The subprotocol 'db2' identifies the IBM DB2 Driver for JDBC and SQLJ, and when used with z/OS, the driver subname specifies a location-name used to identify an instance of the database manager DB2 for z/OS.
- Connection method 2:

195. A connection context class has several different constructors. The following describes using only one of them. For further information see applicable product documentation.

1. Invoke the JDBC `java.sql.DriverManager.getConnection` method.[196] One form of `java.sql.DriverManager.getConnection` takes a single `java.lang.String` that specifies the location name associated with the data source. The invocation returns an instance of class `java.sql.Connection`, which represents a JDBC connection to that data source.
2. For environments other than the CICS environment the default state of autoCommit for a JDBC connection is on. To disable autoCommit, invoke the setAutoCommit method on the Connection object with an argument of `false`.
3. Invoke the constructor for the connection context class. For the argument of the constructor, use the JDBC Connection returned from `java.sql.DriverManager.getConnection`.

For example, with DB2 for z/OS, to use the second method to set up connection context myConn to access the data source associated with location NEWYORK with autoCommit off, execute a connection declaration clause to generate a connection context class:

```
#sql context Ctx;
```

Then register a JDBC driver, and invoke `java.sql.Driver.getConnection` with the argument jdbc:db2:NEWYORK, set autoCommit off for the connection, and invoke the constructor for class Ctx using the JDBC connection as the argument:

```
Class.forName("com.ibm.db2.jcc.DB2Driver");
Connection jdbcConn=DriverManager.getConnection("jdbc:db2:NEWYORK");
jdbcConn.setAutoCommit(false);
Ctx myConn=new Ctx(jdbcConn);
```

Connection method 2 results in SQLJ and JDBC sharing the same connection, and is one that may be taken by an application needing both static and dynamic access to the same data source.

Once a connection context is established, to perform an SQL statement at a data source use one of the following two methods:

- Use an *explicit connection.*

  Specify a connection context, enclosed in square brackets, following the #sql. For example, the following executes an UPDATE statement at the data source associated with connection context myConn:

  ```
  #sql [myConn] {UPDATE DEPARTMENT
        SET MGRNO=:hvmgr WHERE DEPTNO=:hvdeptno};
  ```

- Use a *default connection.*

  When an executable clause does not specify a connection context, a default connection context is used. SQLJ's default connection context is implemented by the class sqlj.runtime.ref.DefaultContext. An application's default connection can be set to a specified data source using the sqlj.runtime.ref.DefaultContext.setDefaultContext method, after which that connection will be used as if it had been explicitly specified by any executable clause that does not specify a connection context. Alternatively, if setDefaultContext is not used to override it the default connection will be to the default relational database.

Use of a default connection context is not recommended. The reasons are as follows. First, the default context is not fully specified by the applicable standards, for example its class name sqlj.runtime.ref.DefaultContext is not defined by

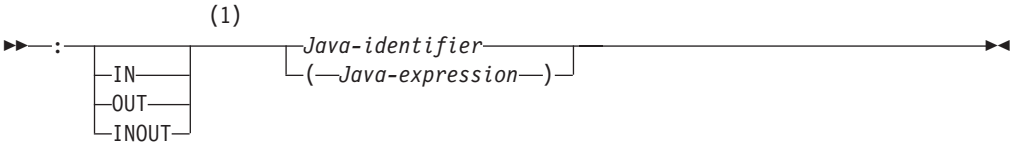
196. DriverManager.getConnection has several different signatures. The following describes using only one of them. For further information see applicable product documentation.

standard, and any reference to it could result in non-portable applications. Second, the setDefaultContext method is implemented using a static variable which may cause difficulties for reentrant or multi-threaded applications. Use of explicit connections is considered safer.

Using host variables and expressions in Java

Use of a host variable or an expression in embedded SQL is similar to using those variables or expressions in any Java statement, and all of the rules for a Java variable being in scope and declared before it is used apply. There is no requirement that a host variable appear in a declare section and SQLJ supports neither the BEGIN DECLARE SECTION nor the END DECLARE SECTION statements.

Syntax



Notes:

1 The Java programming language is case-sensitive and lower case is typically used for keywords. However, keywords used in embedding a host variable or expression, and outside the expression's embedded *Java-expression*, are considered SQL keywords. These keywords are shown in upper case and able to appear in the statement in any mix of upper or lower case.

In an executable clause a simple variable can be referenced by preceding it with a colon (:). A Java expression can be used by enclosing it in parentheses, '()', and preceding the left parenthesis with a colon. For example to update the yearly bonus of the employee identified by the host variable empID, based on an expression involving the host variable yearsEmployed, one might use:

```
#sql {UPDATE EMPLOYEE
      SET BONUS=:(((int) yearsEmployed/5)*500) WHERE EMPNO=:empID};
```

The expression '((int) yearsEmployed/5)*500' is evaluated with Java's rules for rounding and truncation, and including any side effects that would occur had it appeared outside of an executable clause (for example, had it been '((int) yearsEmployed++/5)*500', yearsEmployed would have been incremented following its use), and the expression's result is the value assigned to BONUS. Note that use of an array is treated as use of an expression, and must be enclosed in parentheses. In other words, if 'hArray' is a Java array object then ':hArray[5]' is not properly formed and must instead be specified as ':(hArray[5])'

When invoking a procedure, it may be necessary to indicate whether a host variable or expression represents an IN, OUT, or INOUT parameter, i.e., to specify a parameter's *parameter mode*. This is done by following the introductory colon with the appropriate IN, OUT, or INOUT keyword. If not specified, the parameter mode is assumed to be IN. Parameter modes must be correct for each parameter of the procedure invoked or the necessary code will not be generated to, for example, assign the value of an OUT parameter to its target host variable. Outside a CALL statement, parameter mode has little meaning. If specified for an input value, then

IN may be specified. If parameter mode is specified in a situation where output is involved, for example the INTO portion of a FETCH statement, then OUT may be specified.

Using SQLJ iterators to retrieve rows from a result table

The SQLJ equivalent of a cursor is an *SQLJ iterator*. An SQLJ iterator is defined using an *iterator declaration clause*. An SQLJ iterator is either a *positioned iterator* or a *named iterator*. All iterator declaration clauses specify:

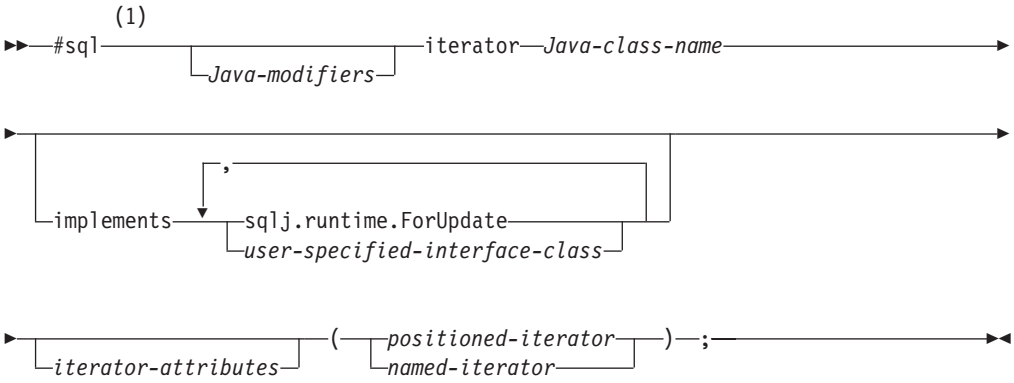
- information for its generated Java class declaration, such as whether the iterator is public[197] or static
- a set of static attributes, in an *iterator-attributes* clause, such as whether the iterator is holdable or whether columns of its underlying table or view can be updated
- a list of Java data types, and in the case of a named iterator the names of the accessor methods to be used to access the columns of the underlying cursor.

As explained in the next sections, whether the named or positioned type of SQLJ iterator is chosen impacts both how an iterator is declared and how an iterator is used.

Declaring iterators

An iterator declaration clause may appear anywhere in a program that a Java class declaration is permitted.

Syntax

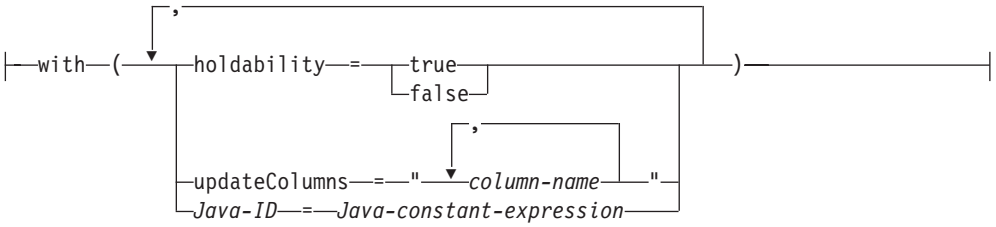


Notes:

1 The Java programming language is case-sensitive and lower case is typically used for keywords. For that reason, unlike other SQL keywords, the keywords appearing in an iterator declaration clause are shown in lower case and must appear in the statement in lower case.

iterator-attributes:

197. Iterators must be public when an *iterator-attributes* clause is specified.



positioned-iterator:

named-iterator:

Description

Java-modifiers
Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
Names the generated iterator class. *Java-class-name* must be a valid Java identifier.

implements
The implements clause specifies one or more user-defined Java interfaces, or the SQLJ interface sqlj.runtime.ForUpdate, that this iterator supports.

Each contained *user-specified-interface-class* must identify a valid Java interface according to Java's rules for use of interfaces. The iterator must be declared to implement at least the SQLJ interface sqlj.runtime.ForUpdate if it is to be referenced in a positioned UPDATE or positioned DELETE operation.

with
Introduces a set of static attributes of the generated iterator class and the initial value of each such static attribute.

holdability
Specifies a Java boolean value that indicates whether an iterator keeps its position in a table after a COMMIT statement is executed.

updateColumns
Lists the *column-name*s of the underlying table or view allowed to be modified when the iterator is used in a positioned UPDATE statement. The value for updateColumns is a Java String literal containing column names, separated by commas.

If updateColumns is specified in a with element of an iterator declaration clause, the iterator declaration clause must contain an implements clause that includes sqlj.runtime.ForUpdate.

Java-ID
Names a user-defined static attribute of a generated iterator class. *Java-ID* must be a valid Java identifier. The value of *Java-constant-expression*, which supplies that attribute's initial value, is also user-defined.

positioned-iterator
Specifies a list of one or more Java data types. These data types describe the columns of the result table, in left-to-right order.

Java-data-type
The Java data type of a column of the result table of a positioned iterator.

named-iterator
Specifies a list of one or more Java data types and Java accessor method identifiers.

Java-data-type
The Java data type of a column of the result table of the named iterator, and the result data type of the accessor method for that column.

column-accessor
Names an accessor method for a column of the result table of the named iterator class. *column-accessor* must be a valid Java identifier.

Using positioned iterators to retrieve rows from a result table

A positioned iterator is the type most like a cursor in non-Java applications. The columns of a positioned iterator correspond to the columns of the result table, in left-to-right order. If an iterator declaration clause contains two or more data type declarations, the first corresponds to the first column in the result table, the second to the second column in the result table, and so on.

When an iterator declaration clause for a positioned iterator is encountered, it is replaced with a generated *positioned iterator class* with the name specified in the iterator declaration clause. An object of the positioned iterator's class can then be used to fetch rows from a result table.

For example, suppose rows are to be retrieved from a result table containing the values of the LASTNAME and HIREDATE columns of the table EMPLOYEE. A positioned iterator class is first declared with two columns of the appropriate data types, see “Determining equivalent SQL and Java data types” on page 1112 for additional information. The following declares the class ByPos, whose first column is of class `String`, and second of JDBC-defined class `java.sql.Date`. It then declares positer to be object of the ByPos class:

```
#sql iterator ByPos(String,java.sql.Date);
ByPos positer;
```

To use an iterator, an *assignment clause* assigns the result table from a SELECT statement to an instance of an iterator class. Figure 14 on page 1108 shows how positer can be used to retrieve the result table rows.

```
    String name = null;
    Date hrdate;
1   #sql positer = {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
2   #sql {FETCH :positer INTO :name, :hrdate};
                              // Retrieve the first row
3   while ( !positer.endFetch() )
    {
      System.out.println(name + " was hired on " + hrdate);
      // Retrieve the next row
      #sql {FETCH :positer INTO :name, :hrdate};
    }
4   positer.close();
```

Figure 14. Retrieving rows using a positioned iterator

Notes to Figure 14:

1 This executable clause performs the SELECT statement, constructs an iterator object containing the result table for the SELECT, and assigns the iterator object to variable positer. In the terminology of other language embeddings this statement performs the functions of both the DECLARE CURSOR and the OPEN statements.

2 The FETCH statement uses left-to-right positional mapping to assign columns of positer's result table to the corresponding variables in the INTO list.

Note that unlike other executable clauses the FETCH statement never needs the iterator's data source to be identified with an explicit connection context. Each instance of an iterator remembers its associated data source.

3 Method endFetch(), a method of the generated iterator class ByPos, returns a value of `true` when all rows have been retrieved from the iterator, and `false` otherwise. The first FETCH statement needs to be executed before endFetch() is called.

4 Method close(), a method of the generated iterator class ByPos, should be called to release resources associated with the iterator when that iterator is no longer needed.

Using named iterators to retrieve rows from a result table

Using named iterators is an alternative way to select rows from a result table. When a named iterator is declared, names are specified that match those of a result table's columns.

When an iterator declaration clause for a named iterator is encountered, it is replaced with a generated *named iterator class* with the name specified in the iterator declaration clause. That generated class includes an accessor method for each column in the iterator declaration clause. The accessor method's name is the name of the column specified in the iterator declaration clause, and its result data type is the data type of the associated column in that clause. As with all Java identifiers an accessor method's name is case sensitive. However, while the accessor method's name is case sensitive, an accessor method's name and a result table column name that differ only in case are considered to be matching names.

The following iterator declaration clause generates the named iterator class ByName, which includes two accessor methods. Those accessor methods are LastName() returning values of class `java.lang.String`, and HireDate() returning values of class `java.sql.Date`. Then nameiter is declared to be an object of the ByName class:

```
#sql iterator ByName(String LastName, java.sql.Date HireDate);
ByName nameiter;
```

To use an iterator, an assignment clause assigns the result table from a SELECT statement to an instance of an iterator class. Figure 15 on page 1109 shows how

nameiter could be used to retrieve rows from a result table containing values of the LASTNAME and HIREDATE columns of the EMPLOYEE table.

```
1  #sql nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
2  while (nameiter.next())
   {
     System.out.println( nameiter.LastName() +
                         " was hired on " + nameiter.HireDate());
   }
3  nameiter.close();
```

Figure 15. Retrieving rows using a named iterator

Notes to Figure 15:

1. This executable clause performs the SELECT statement, constructs an iterator object containing the result table for the SELECT, and assigns the iterator object to variable nameiter. In the terminology of other language embeddings this statement performs the functions of both the DECLARE CURSOR and the OPEN statements.
2. Method next(), a method of the generated class ByName, replaces the FETCH statement of positioned iterators. It advances the iterator to successive rows of the result set. `next` returns a value of `true` when a next row is available, and a value of `false` when all rows have been fetched.
3. Method close(), a method of the generated iterator class ByName, should be called to release resources associated with the iterator when that iterator is no longer needed.

The names of a named iterator's accessor methods must be valid Java identifiers. The names must also match the column names in the result table from which the iterator retrieves its rows. If a SELECT statement that will be assigned to a named iterator involves columns that either have no names or whose names might not be valid Java identifiers, the SQL AS clauses can be used to give columns of the result table acceptable names.

For example, suppose a named iterator is to be used to retrieve the rows specified by this statement:

```
SELECT PUBLIC FROM GOODTABLE
```

The iterator column name must match the column name of the result table, but a name of `public` cannot be specified because `public` is a reserved Java keyword. This leaves one of two choices. First, because Java is case sensitive, the iterator could declare that a name such as puBlic, or PUBlic, or PUBLIC be given to the `PUBLIC` column, or an AS clause could be used to rename `PUBLIC` to a Java identifier that is not similar to a keyword. For example:

```
SELECT PUBLIC AS IS_PUBLIC FROM GOODTABLE
```

A named iterator with a column name that is a valid Java identifier and matches the column name of the result table can then be declared:

```
#sql iterator ByName(String is_public);
ByName nameiter;
```

And nameiter could then be used as the target of an assignment clause:

```
#sql nameiter={SELECT PUBLIC AS IS_PUBLIC FROM GOODTABLE};
```

Using iterators for positioned update and delete operations

When declaring an iterator that will be used in a positioned UPDATE or DELETE statement, an SQLJ *implements clause* is used to specify that the iterator implements the `sqlj.runtime.ForUpdate` interface. The iterator must also be declared as public. For example, suppose instances of iterator class ByPos are to be used in a positioned DELETE statement. The declaration would be:

```
#sql public iterator ByPos(String) implements sqlj.runtime.ForUpdate
  with(updateColumns="EMPNO");
```

Because the iterator is public but not static Java requires that it either be declared in a different source file, or be declared as a nested class. To use the iterator when it is declared in a different source file:

1. Import the generated iterator class.
2. Declare an instance of the generated iterator class.
3. Assign the SELECT statement associated with the positioned UPDATE or DELETE to the iterator instance.
4. Execute positioned UPDATE or DELETE statements using the iterator.

After the iterator is created, any application that has addressability to the iterator and imports the generated class can retrieve data and execute positioned UPDATE or DELETE statements using the iterator. The authorization ID under which a positioned UPDATE or DELETE statement executes is the authorization ID under which the DB2 package containing the UPDATE or DELETE executes.

For example, consider the named iterator UpdByName declared in the following example.

```
#sql public iterator UpdByName(String EMPNO, BigDecimal SALARY)
  implements sqlj.runtime.ForUpdate
  with(updateColumns="SALARY");
```

To use UpdByName for a positioned UPDATE in another file, execute statements like those in Figure 16.

```
1  import UpdByName;
       :
       :
   {
     UpdByName upditer;         // Declare object of UpdByName class
     String enum;
2    #sql upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
                         WHERE WORKDEPT='D11'};
3    while (upditer.next())
     {
       enum = upditer.EMPNO();  // Get value from result table
4      #sql {UPDATE EMPLOYEE SET SALARY=SALARY*1.05 WHERE CURRENT OF :upditer};
                              // Update row where cursor is positioned
       System.out.println("Updating row for " + enum);
     }
     upditer.close();           // Close the iterator
     #sql {COMMIT};             // Commit the changes
   }
```

Figure 16. Updating rows using a positioned iterator

Notes to Figure 16:

1. This statement imports named iterator class UpdByName, generated by the iterator declaration clause for UpdByName in UpdByName.sqlj. The import command is not needed if UpdByName is in the same package as the Java source file that references it.
2. This executable clause performs the SELECT statement, constructs an iterator object containing the result table for the SELECT, and assigns the iterator object to variable updater.
3. This statement positions the iterator to the next row to be updated.
4. This executable clause performs the positioned UPDATE.

Handling SQL errors and warnings in Java

A Java program containing SQL statements does not use an SQLCA or support the WHENEVER statement. SQLJ `throws` an Exception of the JDBC-defined class `java.sql.SQLException` whenever an SQL statement returns an error. To handle SQL errors, import `java.sql.SQLException` and use the Java language `try/catch` blocks to modify program flow when an SQL error is returned. After an exception is caught, the SQLException's `getErrorCode` method can be used to retrieve a return code and its `getSQLState` method to retrieve SQLSTATE values. For example, the following SELECT INTO statement would fail and an SQLException would be thrown if more than one row exists for the employee whose EMPNO is '000010':

```
try
{
  #sql {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
}
catch(SQLException e)
{
  System.out.println("SQLSTATE returned: " + e.getSQLState());
}
```

Unlike errors, warnings returned by SQL do not result in SQLExceptions. The handling of warnings depends on whether the warning is associated with an executable clause or with an SQLJ iterator. In either case, first import `java.sql.SQLWarning`.

To check for a warning associated with an executable clause, after the clause is executed invoke the `getWarnings` method against the execution context associated with that clause. `getWarnings` returns the first warning an SQL statement generates. Subsequent warnings are chained to the first SQLWarning. An execution context can either be explicitly specified in the embedded SQL statement or accessed from the connection context associated with the statement. The following example retrieves an SQLWarning, with execution context ExecCtx specified explicitly:

```
ExecutionContext ExecCtx = new ExecutionContext();
#sql [ExecCtx] {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
SQLWarning sqlWarn = ExecCtx.getWarnings();
if (sqlWarn != null)
   System.out.println("SQLWarning " + sqlWarn);
```

Alternatively, to access the execution context associated with connection context myConn:

```
#sql [myConn]  {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
ExecutionContext ExecCtx = myConn.getExecutionContext();
```

```
SQLWarning sqlWarn = ExecCtx.getWarnings();
if (sqlWarn != null)
   System.out.println("SQLWarning " + sqlWarn);
```

To check for a warning associated with an SQLJ iterator, invoke the generated iterator class's `getWarnings` method against the iterator. To be aware of all warnings, it is necessary for the getWarnings method to be invoked following each fetch operation. The overhead of those invocations should be weighed against the possible benefit of knowing a warning has been reported. It may be useful to test for warnings only if there is corrective action that an application will take following a warning. In that case, then if, for example, an SQLJ iterator has been declared:

```
#sql positer = {SELECT LASTNAME, SALARY FROM EMPLOYEE};
```

Then an application could test for warnings as shown in the following:

```
#sql {FETCH :positer INTO :name, :sal};
while ( !positer.endFetch() )
{
  SQLWarning sqlWarn = positer.getWarnings();
  if (sqlWarn != null)
     System.out.println("SQLWarning " + sqlWarn);
  System.out.println( name + " has base salary " + sal );
  #sql {FETCH :positer INTO :name, :sal};
}
positer.close();
```

Note that the end of data condition for a result set does not cause getWarnings to report a warning.

An important subclass of both `java.sql.SQLException` and `java.sql.SQLWarning` is that of `java.sql.DataTruncation`. A `java.sql.DataTruncation` exception may be thrown when an update operation storing or modifying data causes a data truncation error to be returned. Alternatively, a `java.sql.DataTruncation` may be reported through `getWarnings()` when a truncation takes place reading data from a data source.

The `java.sql.DataTruncation` class supports methods providing information specific to truncation errors or warnings that is not otherwise available through `java.sql.SQLException` and `java.sql.SQLWarning`. For further information see applicable product documentation.

Determining equivalent SQL and Java data types

There is no Java data type whose value, when output by the database manager, is unable to be recognized as having been an SQL NULL. As a result, the SQLTYPE of all host variables use the values that indicate an associated indicator variable. This aspect of SQLJ's runtime is outside the control of the application programmer. The SQLTYPE and SQLLEN information provided below is for consistency with similar tables in other appendices.

Table 92. Java declarations mapped to typical SQL data types

Java Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
short	501	2	SMALLINT
int, java.lang.Integer	497	4	INTEGER
long, java.lang.Long	492	8	BIGINT

Table 92. Java declarations mapped to typical SQL data types (continued)

Java Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
float, java.lang.Float	481	4	REAL
double, java.lang.Double	481	8	DOUBLE
java.math.BigDecimal	996	16	DECFLOAT(34)[1]
java.lang.String	449	length	VARCHAR(length)
byte[][2]	449	length	VARCHAR(length) FOR BIT DATA
java.sql.Clob[3]	409	length	CLOB(length)
java.sql.Blob[3]	405	length	BLOB(length)
java.sql.Date[3, 4]	385	10	CHAR(10)
java.sql.Time[3, 4]	389	8	CHAR(8)
java.sql.Timestamp[3, 4]	393	26	CHAR(26)
java.sql.SQLXML	989	0	CHAR(26)

Note:

1. Each instance of a BigDecimal class has its own precision and scale. On input, the absence of a known, constant, precision and scale prevents directly mapping the parameter to a DECIMAL or NUMERIC data type. For that reason, DECFLOAT is used.
2. Because this data type is equivalent to a DB2 character data type defined as FOR BIT DATA, SQLJ performs no character conversion for data of this type.
3. This class is part of the JDBC API.
4. The specified SQLTYPE indicates that the contents of the fixed length character string are, as appropriate, a DATE, TIME, or TIMESTAMP. When conveying a distinction between types is less important, 461 (the NUL-terminated VARCHAR representation) may be used instead.

The following table can be used to determine the Java data type that is equivalent to a given SQL data type.

Table 93. SQL data types mapped to typical Java declarations

SQL Data Type	Java Data Type	Notes
SMALLINT	short, java.lang.Integer	The java.lang package defines no wrapper class specific to the primitive type short, so java.lang.Integer is used.
INTEGER	int, java.lang.Integer	
BIGINT	long, java.lang.Long	
DECIMAL(p,s) or NUMERIC(p,s)	java.math.BigDecimal	
DECFLOAT	java.math.BigDecimal	
REAL	float, java.lang.Float	
DOUBLE	double, java.lang.Double	
DECFLOAT(16) or DECFLOAT(34)	java.math.BigDecimal	

Table 93. SQL data types mapped to typical Java declarations (continued)

SQL Data Type	Java Data Type	Notes
CHAR(n)	java.lang.String	*n* is a positive integer. The maximum value of *n* is 254. See Table 60 on page 964 for more information.
VARCHAR(n)	java.lang.String	*n* is a positive integer. The maximum value of *n* is 32 672. See Table 60 on page 964 for more information.
CHAR(n) FOR BIT DATA	byte[][1]	*n* is a positive integer. The maximum value of *n* is 254. See Table 60 on page 964 for more information.
VARCHAR(n) FOR BIT DATA	byte[][1]	*n* is a positive integer. The maximum value of *n* is 32 672. See Table 60 on page 964 for more information.
CLOB(n)	java.sql.Clob[2]	*n* is a positive integer. The maximum value of *n* is 2 147 483 647. See Table 60 on page 964 for more information.
GRAPHIC(n)	java.lang.String	*n* is a positive integer. The maximum value of *n* is 127. See Table 60 on page 964 for more information.
VARGRAPHIC(n)	java.lang.String	*n* is a positive integer. The maximum value of *n* is 16 336. See Table 60 on page 964 for more information.
DBCLOB(n)	no exact equivalent	Not supported.
BLOB(n)	java.sql.Blob[2]	*n* is a positive integer. The maximum value of *n* is 2 147 483 647. See Table 60 on page 964 for more information.
DATE	java.sql.Date[2]	
TIME	java.sql.Time[2]	
TIMESTAMP	java.sql.Timestamp[2]	
XML	java.sql.SQLXML	

Note:

1. Because this data type is equivalent to a DB2 character data type defined as FOR BIT DATA, SQLJ performs no character conversion for data of this type.
2. This class is part of the JDBC API.

Example

The following example, using the IBM DB2 Driver for JDBC and SQLJ DB2 for z/OS, solicits the name of a department, obtains the names and phone numbers of all members of that department from the EMPLOYEE table, and presents that information on the screen.

```
package Reports;

import sqlj.runtime.*;
import java.sql.*;
import java.io.*;
import com.ibm.db2.jcc.*;

#sql context CT1x;

public class Summary
{
  #sql static iterator ReportDept(String lastName, String phoneNo);

  /* Names and Phones by Department */
  public static void main (String[] args) // Main entry point
  throws SQLException
  {
    CT1x myConn=null;
    InputStreamReader inStream = new InputStreamReader(System.in);
    char[] inBuffer = new char[10];
    int ii;
    String workDept;
    ReportDept deptSummary = null;  /* iterator used to process the select */

    /* Get a connection from the IBM DB2 Driver for JDBC and SQLJ, with */
    /* autocommit off. For any errors in setup, print a stack trace and exit. */
    try
    {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      myConn=new CT1x("jdbc:db2:NEWYORK", false);
    }
    catch (SQLException e)
    {
      e.printStackTrace();
      return;
    }
    catch (ClassNotFoundException e)
    {
      e.printStackTrace();
      return;
    }

    try
    {
      /* Get the department number to be used in the SELECT statement and */
      /* put into upper case. */
      System.out.println( "Enter a Department number, followed by a <return> ");
      ii = inStream.read(inBuffer, 0, 10);
      inStream.close();
      workDept = (new String(inBuffer)).trim().toUpperCase();

      /* Perform the select */
      #sql [myConn] deptSummary =
         {SELECT LASTNAME,PHONENO FROM EMPLOYEE WHERE WORKDEPT = :workDept};

      System.out.println("Here are the members of Department " + workDept);
      /* For all rows in the result table */
      while (deptSummary.next())
      {
        /* Display name and phone. If employee does not have a phone, */
        /* then display ? */
        if (deptSummary.phoneNo() == null)
          System.out.println( deptSummary.lastName() + "  ?");
        else
          System.out.println( deptSummary.lastName() + "  " +
                              deptSummary.phoneNo());
      }
```

```
      /* Close the cursor and end the logical unit of work */
      deptSummary.close();
      #sql [myConn] {COMMIT};
    }
    catch (SQLException e)
    {
      e.printStackTrace();
      return;
    }
    catch (java.io.IOException)
    {
      e.printStackTrace();
      return;
    }
    finally
    {
      /* whether an error occurred or not, close any created connection */
      if (myConn != null)
        myConn.close();
    }
  }  /* main */
}  /* Summary */
```

Chapter 18. Coding SQL statements in REXX applications

In the HP-UX, Linux, and Solaris environments, REXX is not supported.

SQL is enabled in REXX through the special REXX command EXECSQL, which is used to pass SQL statements to the database manager for processing. [198]

REXX procedures do not have to be preprocessed. At run time, the REXX interpreter passes SQL statements to the database manager for processing.

The SQL/REXX interface supports the following SQL statements:

ALTER [199]
CALL [201]
CLOSE
COMMIT
COMMENT [199]
CREATE [199]
DECLARE CURSOR
DECLARE GLOBAL TEMPORARY TABLE
DELETE [199, 200]
DESCRIBE
DROP [199]
EXECUTE
EXECUTE IMMEDIATE
FETCH
GRANT [199]
INSERT [199, 200]
LOCK TABLE [199]
MERGE [199, 200]
OPEN
PREPARE
REFRESH TABLE [199]
RELEASE SAVEPOINT
RENAME [199]
REVOKE [199]
ROLLBACK
SAVEPOINT
SET ENCRYPTION PASSWORD [199, 200]
SET PATH [199, 200]
SET SCHEMA[199, 200]
UPDATE [199, 200]

The following SQL statements are not supported by the SQL/REXX interface:

ALLOCATE CURSOR
ASSOCIATE LOCATOR
BEGIN DECLARE SECTION
CONNECT
END DECLARE SECTION
FREE LOCATOR
INCLUDE
RELEASE
SELECT INTO
SET CONNECTION
SET CURRENT DEGREE
SET transition-variable
VALUES
VALUES INTO
WHENEVER [202]

198. In DB2 for LUW in the AIX and Windows for 32-bit operating systems environments, the database manager supports REXX through calls to an external function named SQLEXEC. In the examples that follow, where EXECSQL '...' appears, substitute CALL SQLEXEC '...' in these environments.

199. In DB2 for LUW in the AIX and Windows for 32-bit operating systems environments, this statement is supported via either PREPARE followed by EXECUTE, or by EXECUTE IMMEDIATE.

200. These statements cannot be executed directly if they contain host variables; they must be the object of a PREPARE and then an EXECUTE.

201. The CALL statement cannot include host variables or the USING DESCRIPTOR clause.

Defining the SQL communications area in REXX

The fields that make up the SQL Communications Area (SQLCA) are automatically included by the SQL/REXX interface. An INCLUDE SQLCA statement is not required, nor is it allowed. The SQLSTATE or SQLCODE fields of the SQLCA contain SQL return codes. These values are set by the database manager after each SQL statement is executed. An application can check the SQLSTATE or SQLCODE value to determine whether the last SQL statement was successful.

The SQL/REXX interface uses the SQLCA in a manner consistent with the typical SQL usage. (See Chapter 11, "SQLCA (SQL communication area)," on page 981 for more information.) However, the SQL/REXX interface maintains the fields of the SQLCA in separate variables rather than in a contiguous data area. The variables that the SQL/REXX interface maintains for the SQLCA are defined as follows:

SQLCODE
The SQL return code.

SQLERRMC
Error and warning message tokens.

SQLERRP
Product code and, if there is an error, the name of the module that returned the error.

SQLERRD.*n*
Six variables (*n* is a number between 1 and 6) containing diagnostic information.

SQLWARN.*n*
Eleven variables (*n* is a number between 0 and 10) containing warning flags.

SQLSTATE
An SQL return code that indicates the outcome of the most recently executed SQL statement. Portable applications should use the SQLSTATE return code instead of SQLCODE return code.

Defining SQL descriptor areas in REXX

The following statements require an SQLDA:

CALL...USING DESCRIPTOR *descriptor-name*
DESCRIBE *statement-name* INTO *descriptor-name*
EXECUTE...USING DESCRIPTOR *descriptor-name*
FETCH...USING DESCRIPTOR *descriptor-name*
OPEN...USING DESCRIPTOR *descriptor-name*
PREPARE *statement-name* INTO *descriptor-name* ...

Unlike the SQLCA, there can be more than one SQLDA in a procedure, and an SQLDA can have any valid name. Each SQLDA consists of a set of REXX variables with a common stem, where the name of the stem is the *descriptor-name* from the appropriate SQL statement(s). This must be a simple stem; that is, the stem itself must not contain any periods. The SQL/REXX interface automatically provides the fields of the SQLDA for each unique descriptor name. An INCLUDE SQLDA statement is not required, nor is it allowed.

202. See "Handling SQL errors and warnings in REXX" on page 1121 for more information.

The SQL/REXX interface uses the SQLDA in a manner consistent with the typical SQL usage. (See Chapter 12, "SQLDA (SQL descriptor area)," on page 985 for more information.) However, the SQL/REXX interface maintains the fields of the SQLDA in separate variables rather than in a contiguous data area.

The following variables are returned to the application after a DESCRIBE statement or a PREPARE statement that contains an INTO clause:

stem.n.SQLNAME
The name of the *n*th column in the result table.

stem.SQLCCSID
The CCSID of the *n*th column of data.

The following variables must be provided by the application before an OPEN...DESCRIPTOR, a FETCH...DESCRIPTOR, or an EXECUTE...DESCRIPTOR statement. They are returned to the application after a DESCRIBE statement or a PREPARE statement that contains an INTO clause:

stem.SQLD
Number of variable elements that the SQLDA actually contains.

stem.n.SQLTYPE
An integer representing the data type of the *n*th element (for example, the first element is in stem.1.SQLTYPE).

The following data types are not allowed:

400/401
NUL-terminated graphic string

404/405
BLOB

408/409
CLOB

412/413
DBCLOB

460/461
NUL-terminated character string

504/505
DISPLAY SIGN LEADING SEPARATE

960/961
BLOB locator

964/965
CLOB locator

968/969
DBCLOB locator

996/997
Decimal floating point host variable

stem.n.SQLLEN
If SQLTYPE does not indicate a DECIMAL or NUMERIC data type, the maximum length of the data contained in stem.n.SQLDATA.

stem.n.SQLLEN.SQLPRECISION
If the data type is DECIMAL or NUMERIC, this will contain the precision of the number.

stem.n.SQLLEN.SQLSCALE
If the type is DECIMAL or NUMERIC, this will contain the scale of the number.

The following variables must be provided by the application before an EXECUTE...DESCRIPTOR or OPEN...DESCRIPTOR statement, they are returned to the application after a FETCH...DESCRIPTOR statement. They are not used after a DESCRIBE statement or a PREPARE statement that contains an INTO clause:

stem.n.SQLDATA
This contains the input value supplied by the application, or the output value fetched by SQL.

This value is converted to the attributes specified in SQLTYPE, SQLLEN, SQLPRECISION, and SQLSCALE.

stem.n.SQLIND
If the input or output value is null, this will be a negative number.

Embedding SQL statements in REXX

An SQL statement can be placed anywhere a REXX command can be placed.

G In DB2 for z/OS, a CONNECT statement must be executed to connect to a DB2
G subsystem. In other environments, a CONNECT statement is not required.

Each SQL statement in a REXX procedure must begin with EXECSQL (in any combination of uppercase and lowercase letters), followed by either:

- The SQL statement enclosed in single or double quotes, or
- A REXX variable containing the statement. Note that a colon must not precede a REXX variable when it contains an SQL statement.

For example:

```
EXECSQL "COMMIT"
```

is equivalent to:

```
rexxvar = "COMMIT"
EXECSQL rexxvar
```

The command follows normal REXX rules. For example, it can optionally be followed by a semicolon to allow a single line to contain more than one REXX statement. REXX also permits command names to be included within single quotes; for example:

```
'EXECSQL COMMIT'
```

Comments

Neither SQL comments (--) nor REXX comments are allowed in strings representing SQL statements. Otherwise, normal REXX commenting rules are followed.

Continuation of SQL statements

The string containing an SQL statement can be split into several strings on several lines, separated by commas or concatenation operators, according to standard REXX usage.

Including code

Unlike the other host languages, support is not provided for including externally defined statements.

Margins

There are no special margin rules for the SQL/REXX interface.

Names

Any valid REXX name not ending in a period (.) can be used for a host variable.

Do not use host variable names that begin with 'SQL', 'DB2', 'RDI', 'DSN', 'RXSQL', or 'QRW'. These names are reserved for the database manager.

G In DB2 for z/OS, cursor names and statement names are predefined. These
G predefined names must be used in SQL statements that reference cursors and
G prepared statement names, and they must not be used as host variable names. See
G the product documentation for more information.

Nulls

Although the term *null* is used in both REXX and SQL, the term means different things in the two languages. REXX has a null string (a string of length zero) and a null clause (a clause consisting only of blanks and comments). The SQL null value is a special value that is distinct from all nonnull values and denotes the absence of a (nonnull) value.

Statement labels

REXX command statements can be labeled as usual.

Handling SQL errors and warnings in REXX

The WHENEVER statement is not supported by the SQL/REXX interface. Any of the following may be used instead:

- A test of the REXX SQLSTATE or SQLCODE variables after each SQL statement to detect error and warning conditions issued by the database manager, but not for those issued by the SQL/REXX interface.
- A test of the REXX RC variable after each SQL statement to detect error and warning conditions. Each use of the EXECSQL command sets the RC variable to:

  0 Statement completed successfully.

  positive
  A SQL warning occurred.

  negative
  An SQL error occurred

  This can be used to detect errors and warnings issued by either the database
  G manager or by the SQL/REXX interface. The values of RC are product-specific.
  G In DB2 for LUW, the RC variable is not set to a positive value for warnings.

- The REXX SIGNAL ON ERROR and SIGNAL ON FAILURE facilities can be used to detect errors, but not warnings. This is driven by the REXX RC variable.

G In DB2 for LUW, SIGNAL ON ERROR and SIGNAL ON FAILURE cannot be
G used to detect SQL errors.

Isolation level

To use different isolation levels in REXX see the product documentation.

Using host variables in REXX

REXX does not provide for variable declarations. New variables are recognized by their appearance in assignment statements. Therefore, there is no SQL declare section, and the BEGIN DECLARE SECTION and END DECLARE SECTION statements are not supported.

All host variables within an SQL statement must be preceded by a colon (:).

The SQL/REXX interface performs substitution in compound variables before passing statements to the database manager. For example:

```
a = 1
b = 2
EXECSQL 'OPEN c1 USING :x.a.b'
```

will cause the contents of x.1.2 to be passed to SQL.

The following SQL data types are not supported in REXX:

- BIGINT
- DECFLOAT
- BLOB, CLOB, and DBCLOB
- XML

Determining data types of input host variables

All data in REXX is in the form of strings. The data type of input host variables (that is, host variables used in a 'USING host variable' clause in an EXECUTE or OPEN statement) is inferred by the database manager at run-time from the contents of the variable according to Table 94.

These rules define either numeric, character, or graphic values. A numeric value can be used as input to a numeric column of any type. A character value can be used as input to a character column of any type, or to a date, time, or timestamp column. A graphic value can be used as input to a graphic column of any type.

Table 94. Determining Data Types of Host Variables in REXX

Host Variable Contents	Assumed Data Type	SQL Type Code	SQL Type Description
A number with neither decimal point nor exponent. It can have a leading plus or minus sign.	signed integers	496/497	INTEGER

Table 94. Determining Data Types of Host Variables in REXX (continued)

Host Variable Contents	Assumed Data Type	SQL Type Code	SQL Type Description
A number that includes a decimal point, but no exponent, or A number that does not include a decimal point or an exponent and is greater than 2147483647 or smaller than -2147483647. It can have a leading plus or minus sign. *m* is the total number of digits in the number. *n* is the number of digits to the left of the decimal point (if any).	packed decimal	484/485	DECIMAL(*m*,*n*)
A number that is in scientific or engineering notation (that is, followed immediately by an 'E' or 'e', an optional plus or minus sign, and a series of digits). It can have a leading plus or minus sign.	floating point	480/481	DOUBLE PRECISION
A string with leading and trailing apostrophes (') or quotation marks (''), which has length *n* after removing the two delimiters, or A string of length *n* which cannot be recognized as numeric or graphic via other rules in this table.	varying-length character string	448/449	VARCHAR(*n*)
A string with a leading and trailing apostrophe (') or quotation marks ('') preceded by the character 'G', 'g', 'N', or 'n', which contains *n* DBCS characters. [203]	varying-length character string	464/465	VARGRAPHIC(*n*)
undefined variable	variable for which a value has not been assigned	none	Data that is not valid was detected.

The format of output host variables

It is not necessary to determine the data type of an *output host variable* (that is, a host variable used in an 'INTO host variable' clause in a FETCH statement). Output values are assigned to host variables as follows:

- Character values are assigned without leading and trailing apostrophes.
- Graphic values are assigned without a leading G or apostrophe, without a trailing apostrophe, and without shift-out and shift-in characters.
- Numeric values are translated into strings.
- Integer values do not retain any leading zeros. Negative values have a leading minus sign. Positive values do not have a leading plus sign.
- Decimal values retain leading and trailing zeros according to their precision and scale. Negative values have a leading minus sign. Positive values do not have a leading plus sign.
- Floating-point values are in scientific notation, with one digit to the left of the decimal place. The 'E' is in uppercase.

203. In EBCDIC implementations, the byte immediately following the leading apostrophe or quote is a X'0E' shift-out, and the byte immediately preceding the trailing apostrophe or quote is a X'0F' shift-in.

Avoiding REXX conversion

To guarantee that a string is not converted to a number or assumed to be of graphic type, strings can be enclosed in the following: "'". Simply enclosing the string in apostrophes does not work. For example:

```
stringvar = '100'
```

will cause REXX to set the variable *stringvar* to the string of characters 100 (without the apostrophes). This will be evaluated by the SQL/REXX interface as the number 100, and it will be passed to SQL as such.

On the other hand,

```
stringvar = "'"100"'"
```

will cause REXX to set the variable *stringvar* to the string of characters '100' (with the apostrophes). This will be evaluated by the SQL/REXX interface as the string 100, and it will be passed to SQL as such.

Indicator variables in REXX

An indicator variable is an integer. On retrieval, an indicator variable is used to show whether its associated host variable has been assigned a null value. On assignment to a column, a negative indicator variable is used to indicate that a null value should be assigned.

Unlike other languages, a valid value must be specified in the host variable even if its associated indicator variable contains a negative value.

See “References to host variables” on page 116 for more information on using indicator variables.

Example

The following example solicits the name of a department, obtains the names and phone numbers of all members of that department from the EMPLOYEE table, and presents that information on the screen.

```
/* Names and Phones by Department Exec */

/* If there are any nonzero return codes, then branch to the error handler */
     Signal on error

/* Prepare the select statement */
     stmt = 'SELECT LASTNAME, PHONENO FROM EMPLOYEE WHERE WORKDEPT = ?'
     what_stmt = 'PREPARE'
     EXECSQL 'PREPARE stmt_name FROM :stmt'

/* Declare the cursor to be used for reading the result table */
     what_stmt = 'DECLARE'
     EXECSQL 'DECLARE c1 CURSOR FOR stmt_name'

/* Get the department number to be used in the SELECT and put into upper case) */
     Say 'Enter a Department number'
     Parse upper pull dept

/* Find all rows that satisfy the SELECT */
     what_stmt = 'OPEN'
     EXECSQL 'OPEN c1 USING :dept'

/* Turn off the automatic error trap (in order to handle FETCH warnings in-line */
     Signal off error

/* For all rows in the result table */
```

```
     Say 'Here are the members of Department' dept
     Do forever
       /* Fetch the row */
            what_stmt = 'FETCH'
            EXECSQL 'FETCH c1 INTO :name, :phone :phone_ind'
       /* If no more rows, then done */
            If rc <> 0 & sqlcode = 100 then
               Leave
       /* If error then go to error handler */
            If rc <> 0 then
               Signal error
       /* If employee does not have a phone, then set phone to ? */
            If phone_ind < 0 then
               phone = '?'
       /* Display name and phone */
            Say name phone
     End

/* Turn on the automatic error trap again */
     Signal on error

/* Close the cursor and end the logical unit of work */
     what_stmt = 'CLOSE'
     EXECSQL 'CLOSE c1'

     what_stmt = 'COMMIT'
     EXECSQL 'COMMIT'

     Exit 0

Error:  /* Error handler */
     Signal off error
     Say ' '
     Say 'Error accessing EMPLOYEE table'
     Say 'Statement in error was:' what_stmt
     Say 'RC      =' rc
     Say 'SQLCODE =' sqlcode
     Exit rc
```

Applications

Chapter 19. Coding programs for use by external routines

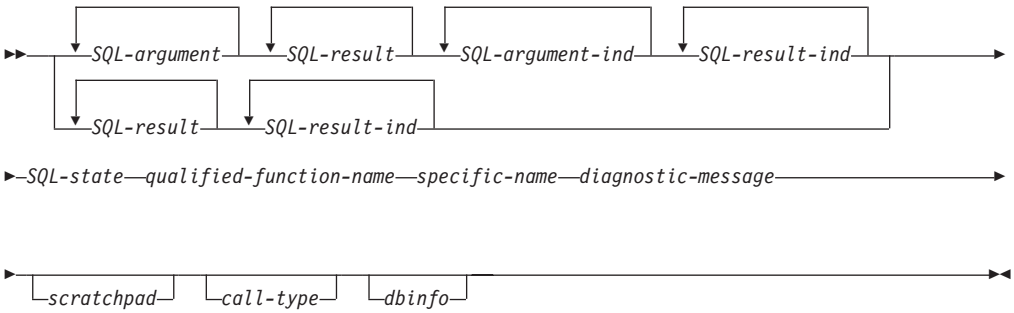
Parameter passing for external routines

An external routine invokes an executable program that must be written to accept parameters according to the specified language and parameter style of the routine.

G Whether the program is written as a main program or a subroutine may be
G specified by the PROGRAM TYPE clause (see "CREATE PROCEDURE (external)" on page 667) or is product specific for the type of routine.
G

Parameter passing for external functions written in C or COBOL

An external scalar function written in C or COBOL must use a parameter style of SQL. An external table function written in C or COBOL must use a parameter style of DB2SQL. When using either parameter style, the database manager passes implicit parameters to the program in addition to the parameters specified in the invocation of the user-defined function. The parameters are passed to the program in the order defined by the following diagram.



SQL-argument
Each *SQL-argument* represents one input parameter defined when the function was created.

Each input parameter of the function is set by the database manager before invoking the program. The value of each of these arguments is taken from the expression specified in the function invocation. It is assigned to the corresponding parameter definition in the CREATE statement using storage assignment as described in "Assignments and comparisons" on page 77. The corresponding parameter is determined by the positional ordering from left to right.

These arguments are input only and any changes to these argument values made by the program are ignored by the database manager upon return from the program.

SQL-result
For a scalar function, *SQL-result* is the output argument of the function, which must be set by the program before returning to the database manager. For a table function, each *SQL-result* represents a column in the result table of the function, which must be set by the program before returning to the database

manager. Each *SQL-result* of a table function corresponds to the column in position from left to right in the RETURNS clause of the routine definition.

If the CAST FROM clause was specified in the CREATE FUNCTION statement, the program is expected to return a data type based on the SQL data type specified immediately following the CAST FROM. Then, the database manager does a second CAST, to the SQL data type specified immediately following the RETURNS. If the CAST FROM clause was not specified in the CREATE FUNCTION statement, the program is expected to return a data type based on the SQL data type specified immediately following the RETURNS keyword.

The program must return a value that corresponds to the data type and length of the result as specified when the function was created. See "Attributes of the arguments of a routine program" on page 1136 for appropriate data type declarations. The *SQL-result* value is assigned to a value with the RETURNS data type or the CAST FROM data type using storage assignment rules as described in "Assignments and comparisons" on page 77.

SQL-argument-ind
There is an *SQL-argument-ind* for each *SQL-argument* passed to the program. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument* and indicates whether the *SQL-argument* has a value or is NULL.

Each *SQL-argument-ind* is defined as a two-byte signed integer.

Each *SQL-argument-ind* associated with an argument of the function is set by the database manager before invoking the program. It contains one of the following values:

0 The argument is present and not NULL.

-1 The argument value is NULL.

-2 The argument value is NULL due to a numeric conversion error (such as divide by 0 or overflow) or a character conversion error.

If the function is defined with RETURNS NULL ON NULL INPUT, the program does not need to check for a NULL value. However, if it is defined with CALLED ON NULL INPUT, any argument can be NULL and the program should check each *SQL-argument-ind*.

SQL-result-ind
For a scalar function, there is an *SQL-result-ind* for the single *SQL-result* of the program, which must be set by the program before returning to the database manager. For a table function, there is an *SQL-result-ind* for each *SQL-result* parameter of the program, which must be set by the program before returning to the database manager. The *n*th *SQL-result-ind* corresponds to the *n*th *SQL-result* of the program.

Each *SQL-result-ind* is defined as a two-byte signed integer.

A result indicator is used by the program to indicate if a result value is NULL:

0 or positive
The result value is present and not NULL.

negative
The result value is NULL.

Any negative value for the indicator set by the program is returned by the database manager as a -1, except for a value of -2 which is returned as a -2.

SQL-result-ind is defined as a two-byte signed integer.

SQL-state
This output argument is a CHAR(5) value that represents the SQLSTATE. This argument is passed in from the database manager with the initial value set to '00000' and can be set by the program as the result SQLSTATE for the function. 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.

qualified-function-name
This input argument is set by the database manager before invoking the program. It is a VARCHAR(517) value that contains the qualified name of the function that is invoking the program. The identifiers in the function name might be delimited. This argument is useful when the program is being used by multiple function definitions so that the program can distinguish which function is being invoked. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

specific-name
This input argument is set by the database manager before invoking the program. It is a VARCHAR(128) value that contains the specific name of the function that is invoking the program. Like *qualified-function-name*, this parameter is useful when the program is being used by multiple function definitions so that the program can distinguish which definition is being invoked. See "CREATE FUNCTION" on page 606 for more information about *specific-name*. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

diagnostic-message
This output argument is a VARCHAR(70) value that can be used by the program to send message text back when an SQLSTATE warning or error is returned by the program. It is initialized by the database manager to an empty string before invoking the program and may be set by the program with descriptive information. The *diagnostic-message* argument value is ignored by the database manager unless the *SQL-state* argument is set by the program to an SQLSTATE class (the first two characters) other than '00'.

scratchpad
This input and output argument is set by the database manager before invoking the program. It is only present if SCRATCHPAD is specified in the CREATE FUNCTION statement. The scratchpad provides the program access to storage that is persistent across function invocations within the same SQL statement.

This argument is a structure with the following elements:

- an INTEGER containing the length of the scratchpad
- the actual scratchpad, initialized to all binary zeroes by the database manager before the first invocation of the program.

The value of the scratchpad is unchanged by the database manager between invocations of program based on iterations of the same function invocation within an SQL statement.

call-type
This input argument is set by the database manager before invoking the program. This argument is present for all external table functions and for an external scalar function if the CREATE FUNCTION statement for the function specified FINAL CALL. The *call-type* argument is an INTEGER value that identifies the type of call. The *call-type* argument is for input only and any changes to the argument value that are made by the program are ignored by the database manager upon return from the program.

For a scalar function, *call-type* contains one of the following values:

SQLUDF_FIRST_CALL (-1)
This is the first invocation of the program for this statement. A first call is a normal call in that all the external function argument values of the parameter style are passed. The scratchpad, if included, is set to binary zeros when the function is invoked with this *call-type*.

SQLUDF_NORMAL_CALL (0)
This is a normal call. All the external function argument values of the parameter style are passed.

SQLUDF_FINAL_CALL (1)
This is a final call. No *SQL-argument* or *SQL-argument-ind* values are passed. The program should not set the *SQL-result*, *SQL-result-ind*, *SQL-state*, or *diagnostic-message* arguments.

SQLUDF_FINAL_CRA (255)
This is a final call that does not allow any SQL statement except the CLOSE statement to be processed. This value for *call-type* is used when the final call is being made as a result of commit or rollback processing. No *SQL-argument* or *SQL-argument-ind* values are passed. This call balances the first call, and can be used to release resources. The program should not set the *SQL-result*, *SQL-result-ind*, *SQL-state*, or *diagnostic-message* arguments.

This value is never used when processing the function on DB2 for i.

For a table function, *call-type* contains one of the following values:

SQLUDF__TF_FIRST (-2)
This is the first invocation of the program for this statement. A first call occurs only if FINAL CALL is specified in the CREATE FUNCTION statement. A first call passes all the external function argument values of the parameter style. The scratchpad, if included, is set to binary zeros when the function is invoked with this *call-type*. The program should not set the *SQL-result* or *SQL-result-ind* arguments for a first call because these parameters are ignored by the database manager upon return from the program.

SQLUDF__TF_OPEN (-1)
This is an open call. All the external function argument values of the parameter style are passed. If the CREATE FUNCTION statement for the function did not specify FINAL CALL, the *scratchpad* (if passed) is initialized. Otherwise, the *scratchpad* is not modified from the first call. The program should not set the *SQL-result* or *SQL-result-ind* arguments for an open call because these parameters are ignored by the database manager upon return from the program.

SQLUDF__TF_FETCH (0)
This is a fetch call. All the external function argument values of the parameter style are passed. The database manager expects a fetch call of a

table function to return either a row comprising the set of returned values or an end-of-table condition indicated by SQLSTATE 02000.

SQLUDF__TF_CLOSE (1)
This is a close call. No *SQL-argument* or *SQL-argument-ind* values are passed. This call balances the open call, and can be used to perform any close processing and to release resources (particularly if there is NO FINAL CALL). In cases such as when the table function is used in a join or a subquery, the sequence of open, fetch, and close calls may be repeated multiple times while processing the higher level query. The program should not set the *SQL-result*, *SQL-result-ind*, *SQL-state*, or *diagnostic-message* arguments.

SQLUDF__TF_FINAL (2)
This is a final call. No *SQL-argument* or *SQL-argument-ind* values are passed. This call balances the first call, and can be used to release resources. The program should not set the *SQL-result*, *SQL-result-ind*, *SQL-state*, or *diagnostic-message* arguments.

SQLUDF__TF_FINAL_CRA (255)
This is a final call that does not allow any SQL statement except the CLOSE statement to be processed. This value for *call-type* is used when the final call is being made as a result of commit or rollback processing. No *SQL-argument* or *SQL-argument-ind* values are passed. This call balances the first call, and can be used to release resources. The program should not set the *SQL-result*, *SQL-result-ind*, *SQL-state*, or *diagnostic-message* arguments.

This value is never used when processing the function on DB2 for i.

dbinfo
This input argument is set by the database manager before invoking the program. It is only present if the CREATE FUNCTION statement for the routine specifies the DBINFO keyword. The argument is a structure whose definition is described in "Database information in external routines (DBINFO)" on page 1138. The *dbinfo* argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

Parameter passing for external functions written in Java

The Java parameter style is the style specified by ISO/IEC FCD 9075-13:2008, *Information technology - Database languages - SQL - Part 13: Java Routines and Types (SQL/JRT)*. When coding a Java method for an external function, the following conventions must be followed.

- The Java method must be a public static method.
- The parameters of the Java method must be a Java type that is equivalent to the SQL data type of the parameter (see Table 95 on page 1136).
- The Java method must return a Java type that is equivalent to the SQL data type of the result defined for the function (see Table 95 on page 1136). The return value is the result of the method.

Consider an example of a function created with parameters of SQL types t1, t2, and t3 and returning type t4 with external name 'jarfile.fname' (jarfile is the Java class name). The database manager will invoke the Java method with the expected Java signature:

```
public static T4 fname (T1 a, T2 b, T3 c) { .....}
```

Where:

- fname is the Java method name
- T1 through T4 are the Java types that correspond to SQL types t1 through t4.
- a, b, and c are arbitrary variable names for the input arguments.

For example, given an external function called sample.test3 that returns INTEGER and takes arguments of type CHAR(5), INTEGER, and DATE, the database manager expects the Java implementation of the function to have the following signature:

```
import java.sql.*;
public class sample
{
  public static int test3(String arg1, int arg2, Date arg3) { ... }
}
```

To return a result of an external function from a Java method when using the JAVA parameter style, simply return the result from the method.

```
{
  ...
  return value;
}
```

SQL NULL values are representable in Java only by variables declared to be instances of a Java class. In such variables, SQL NULL is represented by Java null. The following primitive Java types do not support the SQL NULL value: short, int, long, float, double. If a null value is passed to a parameter that is a Java primitive type, an SQL error is returned.

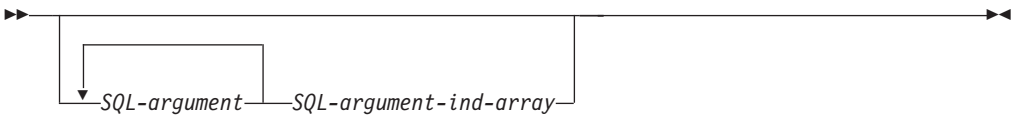
For portability, all Java classes used by an external function must either reside in the jar file installed in the database and referenced in the *external-program-name* of the function definition, or be a class provided by the database manager. If the *external-program-name* does not specify a JAR, then a platform specific mechanism is used to locate classes that are not provided by the database manager.

Parameter passing for external procedures written in C or COBOL

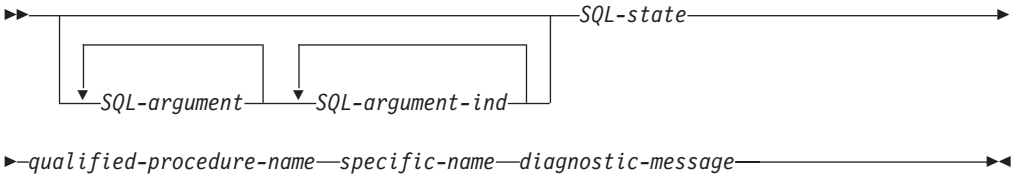
An external procedure written in C or COBOL can be defined to use one of four parameter styles. When using the SQL or DB2SQL parameter style, the database manager passes parameters to the program in addition to the parameters specified in the call to the procedure.Use the SQL parameter style instead of the DB2SQL parameter style unless there is also a need to pass the DBINFO parameter. Depending on the parameter style, the parameters are passed to the program in the order defined by the following diagrams.

Parameter Style GENERAL:

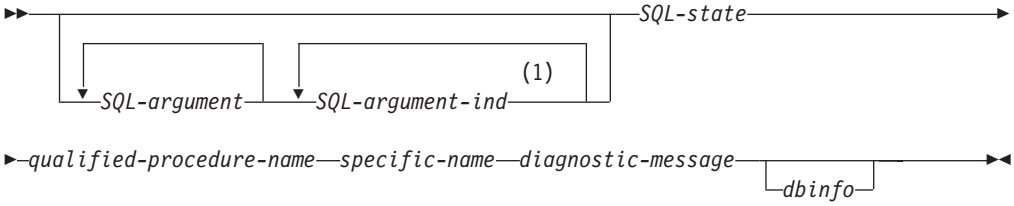
Parameter Style GENERAL WITH NULLS:



Parameter style SQL:



Parameter style DBSQL:



Notes:

G 1 On DB2 for LUW, an *SQL-argument-ind-array* is used.

SQL-argument
Each *SQL-argument* represents one input value, one output value, or both an input value and an output value that is defined when the routine was created.

The following describes the use of each *SQL-argument*.

- An IN parameter of a procedure is set by the database manager before invoking the program. The value of each of these arguments is taken from the expression specified in the CALL to the procedure. It is assigned to the corresponding parameter definition in the CREATE PROCEDURE statement using storage assignment as described in "Assignments and comparisons" on page 77.

  These arguments are input only and any changes to these argument values made by the program are ignored upon return from the program.
- An OUT parameter of a procedure is set by the program before returning to the database manager. The program must return a value that corresponds to the data type and length of the result as specified when the procedure was created. See "Attributes of the arguments of a routine program" on page 1136 for appropriate data type declarations.
- An INOUT parameter of a procedure behaves as both an IN and an OUT parameter and therefore follows both sets of rules described above.

SQL-argument-ind-array
There is an element in *SQL-argument-ind-array* for each *SQL-argument* passed to the program. *SQL-argument-ind-array* is an array of two-byte signed integers. The *n*th element of *SQL-argument-ind-array* corresponds to the *n*th *SQL-argument*. The elements of the array can be used by the program to determine if the corresponding *SQL-argument* is null or not.

The following describes the use of each *SQL-argument-ind-array* element.

- An IN parameter of a procedure is set by DB2 before invoking the program. It contains one of the following values:

  0 The procedure argument is present and not NULL.

  -1 The procedure argument value is NULL.

  -2 The argument value is NULL due to a numeric conversion error (such as divide by 0 or overflow) or a character conversion error.

  The program should check every input argument's *SQL-argument-ind-array* element because any argument can be NULL.
- An OUT parameter of a procedure which must be set by the program before returning to the database manager. This argument is used by the program to indicate if the particular returned value is NULL:

  0 or positive
  The returned value is present and not NULL.

  negative
  The returned value is NULL.

  The program must set the *SQL-argument-ind-array* element of all output parameters. If the indicator value is other than -1 or -2, the returned value may not be the same as the value specified in the program.
- An INOUT parameter of a procedure behaves as both an IN and an OUT parameter and therefore follows both sets of rules described above.

SQL-argument-ind
There is an *SQL-argument-ind* for each *SQL-argument* passed to the program. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument* and indicates whether the *SQL-argument* has a value or is NULL.

Each *SQL-argument-ind* is defined as a two-byte signed integer. The use of each *SQL-argument-ind* is the same as the use of each element of *SQL-argument-ind-array*.

G DB2 for LUW uses an *SQL-argument-ind-array* for parameter style DB2SQL on
G procedures. See the description of *SQL-argument-ind-array* for details.

SQL-state
This output argument is a CHAR(5) value that represents the SQLSTATE. This argument is passed in from the database manager with the initial value set to '00000' and can be set by the program as an SQLSTATE for the procedure. 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.

qualified-procedure-name
This input argument is set by the database manager before invoking the program. It is a VARCHAR(517) value that contains the name of the procedure that is invoking the program. The format of the value in *qualified-procedure-name* is:

```
"schema-name"."procedure-name"
```

Note that any double quote character within the *schema-name* or *procedure-name* gets doubled. This argument is useful when the program is being used by

multiple procedure definitions so that the program can distinguish which procedure is being invoked. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

specific-name
This input argument is set by the database manager before invoking the program. It is a VARCHAR(128) value that contains the specific name of the procedure that is invoking the program. Like *qualified-procedure-name*, this parameter is useful when the routine code is being used by multiple procedure definitions so that the program can distinguish which definition is being invoked. See CREATE PROCEDURE for more information about *specific-name*. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

diagnostic-message
This output argument is a VARCHAR(70) value that can be used by the program to send message text back when an SQLSTATE warning or error is returned by the program. It is initialized by the database manager to an empty string before invoking the program and may be set by the program with descriptive information. The *diagnostic-message* argument value is ignored by the database manager unless the *SQL-state* argument is set by the program to an SQLSTATE class (the first two characters) other than '00'.

dbinfo
This output argument is set by the database manager before invoking the program. It is only present if DBINFO is specified in the CREATE PROCEDURE statement. The argument is a structure whose definition is described in "Database information in external routines (DBINFO)" on page 1138. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

Parameter passing for external procedures written in Java

The Java parameter style is the style specified by ISO/IEC FCD 9075-13:2008, *Information technology - Database languages - SQL - Part 13: Java Routines and Types (SQL/JRT)*. When coding a Java method for an external procedure, the following conventions must be followed.

- The Java method must be a public void static (not instance) method.
- The parameters of the Java method must be a Java type that is equivalent to the SQL data type of the parameter (see Table 95 on page 1136).
- The output parameters must be returned using single element arrays.
- If the procedure is defined with DYNAMIC RESULT SETS *n*, where *n* is greater than zero, the Java method signature must end with *n* parameters whose type is java.sql.ResultSet[ ]. All java.sql.ResultSets to be returned to the calling application must be assigned to the first element of the array representing their output parameter, and all ResultSets that are not being returned to the calling application need to be explicitly or implicitly closed before the procedure returns.

Consider an example of a procedure created with parameters of SQL types t1, t2, and t3, and t4 with external name 'jarfile.pname' (jarfile is the Java class name). The database manager will invoke the Java method with the expected Java signature:

```
public static void pname (T1 a, T2 b, T3 c, T4 d) { ... }
```

Where:

- pname is the Java method name
- T1 through T4 are the Java types that correspond to SQL types t1 through t4
- a, b, c, and d are arbitrary variable names for the arguments.

SQL NULL values are representable in Java only by variables declared to be instances of a Java class. In such variables, SQL NULL is represented by Java null. The following primitive Java types do not support the SQL NULL value: short, int, long, float, double. If a null value is passed to a parameter that is a Java primitive type, an SQL error is returned.

For portability, all Java classes used by an external procedure must either reside in the jar file installed in the database and referenced in the *external-program-name* of the procedure definition, or be a class provided by the database manager. If the *external-program-name* does not specify a JAR, then a platform specific mechanism is used to locate classes that are not provided by the database manager.

Attributes of the arguments of a routine program

Table 95 should be used to determine the appropriate type declarations for the parameters of the program associated with a routine. Each programming language supports different data types. The SQL data type is contained in the leftmost column of the table. Other columns in that row contain an indication of whether that data type is supported as a parameter type for a particular language. If the column contains a dash (-), the data type is not supported as a parameter type for that language.

Table 95. Data type mappings for parameters

SQL Data Type	C and C++	COBOL	Java
SMALLINT	short	PIC S9(4) BINARY	short
INTEGER	sqlint32	PIC S9(9) BINARY	int
BIGINT	sqlint64	PIC S9(18) BINARY	long
DECIMAL(p,s) or NUMERIC(p,s)	-	PIC S9(p-s)V9(s) PACKED-DECIMAL **Note:** Precision must not be greater than 18.	java.math.BigDecimal
REAL or FLOAT(p)	float	COMP-1	float
DOUBLE PRECISION or FLOAT or FLOAT(p)	double	COMP-2	double
DECFLOAT(16) or DECFLOAT(34)	Not supported	Not supported	java.math.BigDecimal
CHARACTER(n)	char ... [n+1]	PIC X(n)	java.lang.String
CHAR(n) FOR BIT DATA	char ... [n+1]	PIC X(n)	byte[]

204. In C, C++, or COBOL, a DATE or TIME value is passed to a routine using a string representation in the ISO format. For example, a TIME value is passed to routine as the string '12.58.01'. When returning a DATE or TIME value, any of the supported DATE or TIME string formats to be used except from a table function where the ISO format must be used. See "String representations of datetime values" on page 66 for details.

Table 95. Data type mappings for parameters (continued)

SQL Data Type	C and C++	COBOL	Java
VARCHAR(n)	char ... [n+1]	Varying-Length Character String (see "Character host variables (excluding CLOB)" on page 1087)	java.lang.String
VARCHAR(n) FOR BIT DATA	VARCHAR structured form (see "Character host variables (excluding CLOB)" on page 1068)	Varying-Length Character String (see "Character host variables (excluding CLOB)" on page 1087.	byte[]
GRAPHIC(n)	sqldbchar ... [n+1]	PIC G(n) DISPLAY-1 or PIC N(n)	java.lang.String
VARGRAPHIC(n)	VARGRAPHIC structured form (see C chapter)	Varying-Length Graphic String (see "Graphic host variables (excluding DBCLOB)" on page 1088)	java.lang.String
DATE[204]	char ... [11]	PIC X(10)	java.sql.Date
TIME[204]	char ... [9]	PIC X(8)	java.sql.Time
TIMESTAMP[204]	char ... [27]	PIC X(26)	java.sql.Timestamp
CLOB	CLOB structured form (see "Declaring a LOB parameter")	CLOB structured form (see "Declaring a LOB parameter")	java.sql.Clob
BLOB	BLOB structured form (see "Declaring a LOB parameter")	BLOB structured form (see "Declaring a LOB parameter")	java.sql.Blob
DBCLOB	DBCLOB structured form (see "Declaring a LOB parameter")	DBCLOB structured form (see "Declaring a LOB parameter")	java.sql.Clob
distinct type	[205]	[205]	[205]
Indicator Variable	short	PIC S9(4) BINARY	Not applicable for Java

Declaring a LOB parameter

The declaration of a LOB parameter for a routine written in C or COBOL requires a structure with a length and data fields.

- For a CLOB or BLOB in C, the following is an example declaration for a CLOB(64K) or BLOB(64K) parameter:

```
struct parm1_t
{
  unsigned long length;
  char          data[65536];
} *parm1;
```

205. A distinct type parameter is passed as the source type of the distinct type. Refer to the source type of the distinct type to determine the appropriate language type.

Taking advantage of definitions in the sqludf include file, the same declaration could also be done as follows:

```
struct sqludf_lob *parm1;
```

- For a DBCLOB in C, the following is an example declaration for a DBCLOB(64K) parameter:

```
struct parm2_t
{
  unsigned long length;
  wchar_t       data[65536];
} parm2;
```

Taking advantage of definitions in the sqludf include file, the same declaration could also be done as follows:

```
struct sqludf_lobg *parm1;
```

- For a CLOB or BLOB in COBOL, the following is an example declaration for a CLOB(64K) or BLOB(64K) parameter:

```
01 LOB-PARM1.
   49 LOB-PARM1-LENGTH PIC 9(9) BINARY.
   49 LOB-PARM1-DATA PIC X(65536).
```

- For a DBCLOB in COBOL, the following is an example declaration for a DBCLOB(64K) parameter:

```
01 DBCLOB-PARM2.
   49 DBCLOB-PARM2-LENGTH PIC 9(9) BINARY.
   49 DBCLOB-PARM2-DATA PIC G(8192) DISPLAY-1.
```

Database information in external routines (DBINFO)

Routines sometimes need access to information about the current server and where the routine is invoked. Table 96 contains a description of the relevant fields of the DBINFO structure which provide such information. Detailed information about the DBINFO structure can be found in the sqludf include file.

Table 96. DBINFO fields

Field	Data Type[206]	Description
Relational database name	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The execution time authorization ID.
Environment CCSID Information	structure (see "DBINFO structure for C" on page 1141 or "DBINFO structure for COBOL" on page 1142)	The CCSID information of the environment. See "CCSID information in DBINFO" on page 1140 for more details.
Schema name	VARCHAR(128)	Schema name of the target table where the function reference is either the right side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement. Otherwise empty (zero length).

206. A data type of VARCHAR(n) in this table implies that there is a 2 byte length field followed by character string data. The character string may not be nul-terminated.

Table 96. DBINFO fields (continued)

Field	Data Type[206]	Description
Table name	VARCHAR(128)	Table name of the target table where the function reference is either the right side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement. Otherwise empty (zero length).
Column name	VARCHAR(128)	Column name of the target column where the function reference is either the right side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement. Otherwise empty (zero length).
Product information	CHAR(8)	Identifies the product on which the routine executes. The information has the form *pppvvrrm*, where: • *ppp* is: DSN for DB2 for z/OS QSQ for DB2 for i SQL for DB2 for LUW • *vv* is a two-digit version identifier such as '09'. • *rr* is a two-digit release identifier such as '01'. • *m* is a one-digit modification level such as '0'. For example, if the server is Version 9 of DB2 for z/OS, the value would be 'DSN09010'.
Platform type	INTEGER	Identifies the operating system on which the program that invokes the routine runs. The value is one of these: • 0 Unknown • 3 Windows • 4 AIX • 5 Windows NT • 6 HP-UX • 7 Solaris • 8 z/OS • 18 Linux • 24 Linux/390 • 25 Linux/zSeries • 26 Linux/IA64 • 27 Linux/PPC • 28 Linux/PPC64 • 29 Linux/X8664 • 30 HP-PA64 • 31 HP-IA • 32 HP-IA64 • 400 i operating system

Table 96. DBINFO fields (continued)

Field	Data Type[206]	Description
Number of column list entries	SMALLINT	For table functions only, the number of entries in the column list array.
Table function column list	pointer to 2-byte integer array	For table functions only, an array corresponding to the ordinal numbers of the columns that this invocation requires from the function. The pointer is null if the function is not a table function. For more details, refer to "Table function column list information in DBINFO."
Application identifier	pointer to character string	If not a null pointer, a pointer to a character string that uniquely identifies the application's connection to the database. A different value is generated for each connection to the database.

CCSID information in DBINFO

The environment CCSID information provided in DBINFO is presented in the form of 3 sets of 3 CCSIDs. Each set consists of an SBCS CCSID, a DBCS CCSID, and a mixed CCSID. The reason for 3 sets of CCSIDs is to allow representations of the different encoding schemes that are possible. Therefore the field following the sets of CCSIDs indicates which set is relevant. The environment CCSIDs provide the routine with information about the CCSID that is used See "Coded character sets and CCSIDs" on page 35 for more information on CCSIDs and codepages.

The meaning of these environment CCSIDs depends on the application server where the routine is executed.

- On DB2 for z/OS, the environment CCSIDs are the CCSIDs associated with the encoding scheme in effect.
- On DB2 for i, the environment CCSIDs are the CCSIDs associated with the job.
- On DB2 for LUW, the environment CCSIDs are the CCSIDs for the relational database.

Table function column list information in DBINFO

If the DBINFO structure is passed to a table function, information about the columns that are required by the caller is passed in the column list array. The database manager allocates the array of 2-byte integers and provides the pointer. If a function is not defined to return a table, this pointer is null. Values are assigned to the elements of the array up to the number of column list entries specified in the DBINFO (referred to as *numtfcol*). The value *numtfcol* is greater than or equal to 0 and less than or equal to the number result columns defined for the user-defined function in the RETURNS TABLE clause of the CREATE FUNCTION statement. The values correspond to the numbers of the columns that the invoking 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 can be in any order. If *numtfcol* is equal to 0, the contents of the array should be ignored. This is the case for a statement like the following one, where the invoking statement needs no column values.

```
SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ
```

This array represents an opportunity for optimization. The user-defined function does not need to return all values for all the result columns of the table function.

Instead, the user-defined function can return only those columns that are needed in the particular context, which you identify by number in the array. However, if this optimization complicates the user-defined function logic enough to cancel the perfomance benefit, you might choose to return every defined column.

DBINFO structure for C

In C, the DBINFO structure and associated structure declarations are equivalent (but not necessarily identical) to the following structure declarations. Overall size of the structure and size of reserved fields is platform specific.

```
#define   SQLUDF_MAX_IDENT_LEN    128    /* max length of identifier */
#define   SQLUDF_SH_IDENT_LEN       8    /* length of short identifier */

/*-------------------------------------------------------------------*/
/* Structure used for: Environment CCSID                             */
/*-------------------------------------------------------------------*/

SQL_STRUCTURE db2_cdpg
{
    struct db2_ccsids
    {
       unsigned long   db2_sbcs;
       unsigned long   db2_dbcs;
       unsigned long   db2_mixed;
    } db2_ccsids_t[3];

   unsigned long   db2_encoding_scheme;
   unsigned char reserved[8];
};

union db_cdpg
{
   /* union includes other platform-specific deprecated structures           */
   /* not included here.                                                     */
   struct db2_cdpg cdpg_db2;           /* Common environment CCSID structure  */
};

/*-------------------------------------------------------------------*/
/* encoding_scheme values for db2_cdpg.db2_encoding_scheme       */
/*-------------------------------------------------------------------*/
#define   SQLUDF_ASCII             0       /* ASCII              */
#define   SQLUDF_EBCDIC            1       /* EBCDIC             */
#define   SQLUDF_UNICODE           2       /* UNICODE            */

/*-------------------------------------------------------------*/
/* Structure used for: dbinfo.                                 */
/*-------------------------------------------------------------*/
SQL_STRUCTURE  sqludf_dbinfo
{
  unsigned short    dbnamelen;                    /* database name length    */
  unsigned char     dbname[SQLUDF_MAX_IDENT_LEN];  /* database name           */
  unsigned short    authidlen;                    /* authorization ID length */
  unsigned char     authid[SQLUDF_MAX_IDENT_LEN];  /* appl authorization ID   */
  union db_cdpg     codepg;                       /* database code page      */
  unsigned short    tbschemalen;                  /* table schema name length*/
  unsigned char     tbschema[SQLUDF_MAX_IDENT_LEN];/* table schema name       */
  unsigned short    tbnamelen;                    /* table name length       */
  unsigned char     tbname[SQLUDF_MAX_IDENT_LEN];  /* table name              */
  unsigned short    colnamelen;                   /* column name length      */
  unsigned char     colname[SQLUDF_MAX_IDENT_LEN]; /* column name             */
  unsigned char     ver_rel[SQLUDF_SH_IDENT_LEN];  /* product information     */
  unsigned long     platform;                     /* platform type           */
  unsigned short    numtfcol;                     /* number of entries in    */
                                                  /* the TF column list array*/
  unsigned char     resd1[24];                    /* Reserved- for expansion */
```

```
  unsigned short   *tfcolumn;                    /* tfcolumn is allocated   */
                                                 /* dynamically if TF is    */
                                                 /* defined; otherwise, this*/
                                                 /* will be a null          */
   char             *appl_id;                     /* application identifier  */
  unsigned char     resd2[20];                   /* Reserved- for expansion */
};
```

DBINFO structure for COBOL

In COBOL, the DBINFO structure and associated structure declarations are equivalent (but not necessarily identical) to the following.

```
      ******************************************************************************
      * Structure used for: dbinfo.
      ******************************************************************************
       01 SQLUDF-DBINFO.
      * relational database name length
           05 DBNAMELEN              PIC 9(4) USAGE BINARY.
      * reltional database name
           05 DBNAME                 PIC X(128).
      * authorization ID length
           05 AUTHIDLEN              PIC 9(4) USAGE BINARY.
      * authorization ID
           05 AUTHID                 PIC X(128).
      * environment CCSID information
           05 CODEPG                 PIC X(48).
           05 CDPG-DB2 REDEFINES CODEPG.
              10 DB2-CCSIDS OCCURS 3 TIMES.
                 15 DB2-SBCS         PIC 9(9) USAGE BINARY.
                 15 DB2-DBCS         PIC 9(9) USAGE BINARY.
                 15 DB2-MIXED        PIC 9(9) USAGE BINARY.
              10 ENCODING-SCHEME     PIC 9(9) USAGE BINARY.
              10 RESERVED            PIC X(8).
      * other platform-specific deprecated CCSID structures not included here
      * schema name length
           05 TBSCHEMALEN            PIC 9(4) USAGE BINARY.
      * schema name
           05 TBSCHEMA               PIC X(128).
      * table name length
           05 TBNAMELEN              PIC 9(4) USAGE BINARY.
      * table name
           05 TBNAME                 PIC X(128).
      * column name length
           05 COLNAMELEN             PIC 9(4) USAGE BINARY.
      * column name
           05 COLNAME                PIC X(128).
      * product information
           05 VER-REL                PIC X(8).
      * Reserved for expansion
           05 RESD0                  PIC X(2).
      * platform type
           05 PLATFORM               PIC 9(9) USAGE BINARY.
      * number of entries in the TF column list array (tfcolumn, below)
           05 NUMTFCOL               PIC 9(4) USAGE BINARY.
      * Reserved for expansion
           05 RESD1                  PIC X(24).
      * tfcolumn will be allocated dynamically if TF is defined; otherwise,
      * this will be a null pointer.
           05 TFCOLUMN               USAGE IS POINTER.
      * application identifier
           05 APPL-ID                USAGE IS POINTER.
      * Reserved for expansion
           05 RESD2                  PIC X(20).
```

Scratch pad in external functions

External functions may need an area to save information between invocations. This is referred to as a *scratch pad*. A function is enabled to have a scratch pad by specifying the SCRATCHPAD keyword during CREATE FUNCTION (see "CREATE FUNCTION (external scalar)" on page 610). Table 97 contains a description of the fields of the scratchpad structure. Detailed information about the scratchpad structure can be found in the sqludf include file.

Table 97. SCRATCHPAD fields

Field	Data Type	Description
Length of scratchpad	INTEGER	Length of the data field of the scratchpad.
Scratchpad area	CHARACTER(100)	The data area available for a scratchpad. Actual length of the scratchpad can exceed 100, but the structure definition in the sqludf include file defaults to 100.

The following is an example of a C declaration for a scratchpad with 150 bytes.

```
SQL_STRUCTURE  sqludf_scratchpad
{
  unsigned long  length;       /* length of scratchpad data        */
  char           data[150];    /* scratchpad data, init. to all \0 */
};
```

Chapter 20. Sample tables

The tables on the following pages are used in the examples that appear throughout this book. This appendix contains the following sample tables:

In these tables, a question mark (?) indicates a null value.

ACT

Name:	ACTNO	ACTKWD	ACTDESC
Type:	SMALLINT NOT NULL	CHAR(6) NOT NULL	VARCHAR(20) NOT NULL
Desc:	Account number	Account keyword	Account description
Values:	10	MANAGE	MANAGE/ADVISE
	20	ECOST	ESTIMATE COST
	30	DEFINE	DEFINE SPECS
	40	LEADPR	LEAD PROGRAM/DESIGN
	50	SPECS	WRITE SPECS
	60	LOGIC	DESCRIBE LOGIC
	70	CODE	CODE PROGRAMS
	80	TEST	TEST PROGRAMS
	90	ADMQS	ADM QUERY SYSTEM
	100	TEACH	TEACH CLASSES
	110	COURSE	DEVELOP COURSES
	120	STAFF	PERS AND STAFFING
	130	OPERAT	OPER COMPUTER SYS
	140	MAINT	MAINT SOFTWARE SYS
	150	ADMSYS	ADM OPERATING SYS
	160	ADMDB	ADM DATA BASES
	170	ADMDC	ADM DATA COMM
	180	DOC	DOCUMENT

CL_SCHED

Name:	CLASS_CODE	DAY	STARTING	ENDING
Type:	CHAR(7)	SMALLINT	TIME	TIME
Desc:	Class code (room:teacher)	Day # of 4 day schedule	Class start time	Class end time
Values:	042:BF	4	12:10 PM	02:00 PM
	553:MJA	1	10:30 AM	11:00 AM
	543:CWM	3	09:10 AM	10:30 AM
	778:RES	2	12:10 PM	02:00 PM
	044:HD	3	05:12 PM	06:00 PM

DEPARTMENT

Name:	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
Type:	CHAR(3) NOT NULL	VARCHAR(29) NOT NULL	CHAR(6)	CHAR(3) NOT NULL	CHAR(16)
Desc:	Department number	Name describing general activities of department	Employee number (EMPNO) of department manager	Department (DEPTNO) to which this department reports	Name of the remote location
Values:	A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	?
	B01	PLANNING	000020	A00	?
	C01	INFORMATION CENTER	000030	A00	?
	D01	DEVELOPMENT CENTER	?	A00	?
	D11	MANUFACTURING SYSTEMS	000060	D01	?
	D21	ADMINISTRATION SYSTEMS	000070	D01	?
	E01	SUPPORT SERVICES	000050	A00	?
	E11	OPERATIONS	000090	E01	?
	E21	SOFTWARE SUPPORT	000100	E01	?

EMP_PHOTO

Name:	EMPNO	PHOTO_FORMAT	PICTURE
Type:	CHAR(6) NOT NULL	VARCHAR(10) NOT NULL	BLOB(100K)
Desc:	Employee number	Photograph format	Photograph
Values:	000130	bitmap	db200130
	000130	gif	db200130
	000140	bitmap	db2000140
	000140	gif	db2000140
	000150	bitmap	db2000150
	000150	gif	db2000150
	000190	bitmap	db2000190
	000190	gif	db2000190

EMP_RESUME

Name:	EMPNO	RESUME_FORMAT	RESUME
Type:	CHAR(6) NOT NULL	VARCHAR(10) NOT NULL	CLOB(5K)
Desc:	Employee number	Resume format	Resume
Values:	000130	ascii	db200130

Name:	EMPNO	RESUME_FORMAT	RESUME
	000130	html	db200130
	000140	ascii	db2000140
	000140	html	db2000140
	000150	ascii	db2000150
	000150	html	db2000150
	000190	ascii	db2000190
	000190	html	db2000190

EMPLOYEE

Names:	EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
Type:	CHAR(6) NOT NULL	VARCHAR(12) NOT NULL	CHAR(1) NOT NULL	VARCHAR(15) NOT NULL	CHAR(3)	CHAR(4)	DATE
Desc:	Employee number	First name	Middle initial	Last name	Department (DEPTNO) in which the employee works	Phone number	Date of hire

JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
CHAR(8)	SMALLINT NOT NULL	CHAR(1)	DATE	DECIMAL(9,2)	DECIMAL(9,2)	DECIMAL(9,2)
Job	Number of years of formal education	Sex (M male, F female)	Date of birth	Yearly salary	Yearly bonus	Yearly commission

EMPNO	FIRSTNME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	ED LEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN		O'CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100

EMPNO	FIRSTNME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	ED LEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FILEREP	16	M	1932-08-11	19950	400	1596
000330	WING		LEE	E21	2103	1976-02-23	FILEREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FILEREP	16	M	1926-05-17	23840	500	1907

EMPPROJACT

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
Type:	CHAR(6) NOT NULL	CHAR(6) NOT NULL	SMALLINT NOT NULL	DECIMAL(5,2)	DATE	DATE
Desc:	Employee number	Project number	Activity number	Proportion of employee's full time to be spent on project	Date activity starts	Date activity ends
Values:	000010	AD3100	10	.50	1982-01-01	1982-07-01
	000070	AD3110	10	1.00	1982-01-01	1983-02-01
	000230	AD3111	60	1.00	1982-01-01	1982-03-15
	000230	AD3111	60	.50	1982-03-15	1982-04-15
	000230	AD3111	70	.50	1982-03-15	1982-10-15
	000230	AD3111	80	.50	1982-04-15	1982-10-15
	000230	AD3111	180	.50	1982-10-15	1983-01-01
	000240	AD3111	70	1.00	1982-02-15	1982-09-15
	000240	AD3111	80	1.00	1982-09-15	1983-01-01
	000250	AD3112	60	1.00	1982-01-01	1982-02-01
	000250	AD3112	60	.50	1982-02-01	1982-03-15
	000250	AD3112	60	1.00	1983-01-01	1983-02-01
	000250	AD3112	70	.50	1982-02-01	1982-03-15
	000250	AD3112	70	1.00	1982-03-15	1982-08-15
	000250	AD3112	70	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.50	1982-10-15	1982-12-01
	000250	AD3112	180	.50	1982-08-15	1983-01-01
	000260	AD3113	70	.50	1982-06-15	1982-07-01
	000260	AD3113	70	1.00	1982-07-01	1983-02-01
	000260	AD3113	80	1.00	1982-01-01	1982-03-01
	000260	AD3113	80	.50	1982-03-01	1982-04-15
	000260	AD3113	180	.50	1982-03-01	1982-04-15
	000260	AD3113	180	1.00	1982-04-15	1982-06-01
	000260	AD3113	180	1.00	1982-06-01	1982-07-01
	000270	AD3113	60	.50	1982-03-01	1982-04-01
	000270	AD3113	60	1.00	1982-04-01	1982-09-01
	000270	AD3113	60	.25	1982-09-01	1982-10-15
	000270	AD3113	70	.75	1982-09-01	1982-10-15
	000270	AD3113	70	1.00	1982-10-15	1983-02-01
	000270	AD3113	80	1.00	1982-01-01	1982-03-01
	000270	AD3113	80	.50	1982-03-01	1982-04-01
	000030	IF1000	10	.50	1982-06-01	1983-01-01
	000130	IF1000	90	1.00	1982-10-01	1983-01-01
	000130	IF1000	100	.50	1982-10-01	1983-01-01

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	000140	IF1000	90	.50	1982-10-01	1983-01-01
	000030	IF2000	10	.50	1982-01-01	1983-01-01
	000140	IF2000	100	1.00	1982-01-01	1982-03-01
	000140	IF2000	100	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-10-01	1983-01-01
	000010	MA2100	10	.50	1982-01-01	1982-11-01
	000110	MA2100	20	1.00	1982-01-01	1983-03-01
	000010	MA2110	10	1.00	1982-01-01	1983-02-01
	000200	MA2111	50	1.00	1982-01-01	1982-06-15
	000200	MA2111	60	1.00	1982-06-15	1983-02-01
	000220	MA2111	40	1.00	1982-01-01	1983-02-01
	000150	MA2112	60	1.00	1982-01-01	1982-07-15
	000150	MA2112	180	1.00	1982-07-15	1983-02-01
	000170	MA2112	60	1.00	1982-01-01	1983-06-01
	000170	MA2112	70	1.00	1982-06-01	1983-02-01
	000190	MA2112	70	1.00	1982-01-01	1982-10-01
	000190	MA2112	80	1.00	1982-10-01	1983-10-01
	000160	MA2113	60	1.00	1982-07-15	1983-02-01
	000170	MA2113	80	1.00	1982-01-01	1983-02-01
	000180	MA2113	70	1.00	1982-04-01	1982-06-15
	000210	MA2113	80	.50	1982-10-01	1983-02-01
	000210	MA2113	180	.50	1982-10-01	1983-02-01
	000050	OP1000	10	.25	1982-01-01	1983-02-01
	000090	OP1010	10	1.00	1982-01-01	1983-02-01
	000280	OP1010	130	1.00	1982-01-01	1983-02-01
	000290	OP1010	130	1.00	1982-01-01	1983-02-01
	000300	OP1010	130	1.00	1982-01-01	1983-02-01
	000310	OP1010	130	1.00	1982-01-01	1983-02-01
	000050	OP2010	10	.75	1982-01-01	1983-02-01
	000100	OP2010	10	1.00	1982-01-01	1983-02-01
	000320	OP2011	140	.75	1982-01-01	1983-02-01
	000320	OP2011	150	.25	1982-01-01	1983-02-01
	000330	OP2012	140	.25	1982-01-01	1983-02-01
	000330	OP2012	160	.75	1982-01-01	1983-02-01
	000340	OP2013	140	.50	1982-01-01	1983-02-01
	000340	OP2013	170	.50	1982-01-01	1983-02-01
	000020	PL2100	30	1.00	1982-01-01	1982-09-15

IN_TRAY

Name:	RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
Type:	TIMESTAMP	CHAR(8)	CHAR(64)	VARCHAR(3000)
Desc:	Date and time received	User id of person sending note	Brief description	The note

Name:	RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
Values:	1988-12-25-17.12.30.000000	BADAMSON	FWD: Fantastic year! 4th Quarter Bonus.	To: JWALKER Cc: QUINTANA, NICHOLLS Jim, Looks like our hard work has paid off. I have some good beer in the fridge if you want to come over to celebrate a bit. Delores and Heather, are you interested as well? Bruce Subject: FWD: Fantastic year! 4th Quarter Bonus. To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas
	1988-12-23-08.53.58.000000	ISTERN	FWD: Fantastic year! 4th Quarter Bonus.	To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas
	1988-12-22-14.07.21.136421	CHAAS	Fantastic year! 4th Quarter Bonus.	To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas

ORG

Name:	DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
Type:	SMALLINT NOT NULL	VARCHAR(14)	SMALLINT	VARCHAR(10)	VARCHAR(13)
Desc:	Department number	Department name	Manager number	Division	Location
Values:	10	Head Office	160	Corporate	New York
	15	New England	50	Eastern	Boston
	20	Mid Atlantic	10	Eastern	Washington

Name:	DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
	38	South Atlantic	30	Eastern	Atlanta
	42	Great Lakes	100	Midwest	Chicago
	51	Plains	140	Midwest	Dallas
	66	Pacific	270	Western	San Francisco
	84	Mountain	290	Western	Denver

PROJACT

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
Type:	CHAR(6) NOT NULL	SMALLINT NOT NULL	DECIMAL(5,2)	DATE NOT NULL	DATE
Desc:	Project number	Activity number	Activity staffing	Activity start	Activity end
Values:	AD3100	10	?	1982-01-01	?
	AD3110	10	?	1982-01-01	?
	AD3111	60	?	1982-01-01	?
	AD3111	60	?	1982-03-15	?
	AD3111	70	?	1982-03-15	?
	AD3111	80	?	1982-04-15	?
	AD3111	180	?	1982-10-15	?
	AD3111	70	?	1982-02-15	?
	AD3111	80	?	1982-09-15	?
	AD3112	60	?	1982-01-01	?
	AD3112	60	?	1982-02-01	?
	AD3112	60	?	1983-01-01	?
	AD3112	70	?	1982-02-01	?
	AD3112	70	?	1982-03-15	?
	AD3112	70	?	1982-08-15	?
	AD3112	80	?	1982-08-15	?
	AD3112	80	?	1982-10-15	?
	AD3112	180	?	1982-08-15	?
	AD3113	70	?	1982-06-15	?
	AD3113	70	?	1982-07-01	?
	AD3113	80	?	1982-01-01	?
	AD3113	80	?	1982-03-01	?
	AD3113	180	?	1982-03-01	?
	AD3113	180	?	1982-04-15	?
	AD3113	180	?	1982-06-01	?
	AD3113	60	?	1982-03-01	?
	AD3113	60	?	1982-04-01	?
	AD3113	60	?	1982-09-01	?
	AD3113	70	?	1982-09-01	?
	AD3113	70	?	1982-10-15	?
	IF1000	10	?	1982-06-01	?
	IF1000	90	?	1982-10-01	?
	IF1000	100	?	1982-10-01	?
	IF2000	10	?	1982-01-01	?
	IF2000	100	?	1982-01-01	?
	IF2000	100	?	1982-03-01	?
	IF2000	110	?	1982-03-01	?
	IF2000	110	?	1982-10-01	?
	MA2100	10	?	1982-01-01	?

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
	MA2100	20	?	1982-01-01	?
	MA2110	10	?	1982-01-01	?
	MA2111	50	?	1982-01-01	?
	MA2111	60	?	1982-06-15	?
	MA2111	40	?	1982-01-01	?
	MA2112	60	?	1982-01-01	?
	MA2112	180	?	1982-07-15	?
	MA2112	70	?	1982-06-01	?
	MA2112	70	?	1982-01-01	?
	MA2112	80	?	1982-10-01	?
	MA2113	60	?	1982-07-15	?
	MA2113	80	?	1982-01-01	?
	MA2113	70	?	1982-04-01	?
	MA2113	80	?	1982-10-01	?
	MA2113	180	?	1982-10-01	?
	OP1000	10	?	1982-01-01	?
	OP1010	10	?	1982-01-01	?
	OP1010	130	?	1982-01-01	?
	OP2010	10	?	1982-01-01	?
	OP2011	140	?	1982-01-01	?
	OP2011	150	?	1982-01-01	?
	OP2012	140	?	1982-01-01	?
	OP2012	160	?	1982-01-01	?
	OP2013	140	?	1982-01-01	?
	OP2013	170	?	1982-01-01	?
	PL2100	30	?	1982-01-01	?

PROJECT

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
Type:	CHAR(6) NOT NULL	VARCHAR(24) NOT NULL	CHAR(3) NOT NULL	CHAR(6) NOT NULL	DECIMAL(5,2)	DATE	DATE	CHAR(6)
Desc:	Project number	Project name	Department responsible	Employee responsible	Estimated mean staffing	Estimated start date	Estimated end date	Major project, for a subproject
Values:	AD3100	ADMIN SERVICES	D01	000010	6.5	1982-01-01	1983-02-01	?
	AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	1982-01-01	1983-02-01	AD3100
	AD3111	PAYROLL PROGRAMMING	D21	000230	2	1982-01-01	1983-02-01	AD3110
	AD3112	PERSONNEL PROGRAMMING	D21	000250	1	1982-01-01	1983-02-01	AD3110
	AD3113	ACCOUNT PROGRAMMING	D21	000270	2	1982-01-01	1983-02-01	AD3110
	IF1000	QUERY SERVICES	C01	000030	2	1982-01-01	1983-02-01	?
	IF2000	USER EDUCATION	C01	000030	1	1982-01-01	1983-02-01	?
	MA2100	WELD LINE AUTOMATION	D01	000010	12	1982-01-01	1983-02-01	?
	MA2110	W L PROGRAMMING	D11	000060	9	1982-01-01	1983-02-01	MA2100
	MA2111	W L PROGRAM DESIGN	D11	000220	2	1982-01-01	1982-12-01	MA2110

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
	MA2112	W L ROBOT DESIGN	D11	000150	3	1982-01-01	1982-12-01	MA2110
	MA2113	W L PROD CONT PROGS	D11	000160	3	1982-02-15	1982-12-01	MA2110
	OP1000	OPERATION SUPPORT	E01	000050	6	1982-01-01	1983-02-01	?
	OP1010	OPERATION	E11	000090	5	1982-01-01	1983-02-01	OP1000
	OP2000	GEN SYSTEMS SERVICES	E01	000050	5	1982-01-01	1983-02-01	?
	OP2010	SYSTEMS SUPPORT	E21	000100	4	1982-01-01	1983-02-01	OP2000
	OP2011	SCP SYSTEMS SUPPORT	E21	000320	1	1982-01-01	1983-02-01	OP2010
	OP2012	APPLICATIONS SUPPORT	E21	000330	1	1982-01-01	1983-02-01	OP2010
	OP2013	DB/DC SUPPORT	E21	000340	1	1982-01-01	1983-02-01	OP2010
	PL2100	WELD LINE PLANNING	B01	000020	1	1982-01-01	1982-09-15	MA2100

SALES

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
Type:	DATE	VARCHAR(15)	VARCHAR(15)	INTEGER
Desc:	Date	Sales person	Region	Sales amount
Values:	1995-12-31	LUCCHESSI	Ontario-South	1
	1995-12-31	LEE	Ontario-South	3
	1995-12-31	LEE	Quebec	1
	1995-12-31	LEE	Manitoba	2
	1995-12-31	GOUNOT	Quebec	1
	1996-03-29	LUCCHESSI	Ontario-South	3
	1996-03-29	LUCCHESSI	Quebec	1
	1996-03-29	LEE	Ontario-South	2
	1996-03-29	LEE	Ontario-North	2
	1996-03-29	LEE	Quebec	3
	1996-03-29	LEE	Manitoba	5
	1996-03-29	GOUNOT	Ontario-South	3
	1996-03-29	GOUNOT	Quebec	1
	1996-03-29	GOUNOT	Manitoba	7
	1996-03-30	LUCCHESSI	Ontario-South	1
	1996-03-30	LUCCHESSI	Quebec	2
	1996-03-30	LUCCHESSI	Manitoba	1
	1996-03-30	LEE	Ontario-South	7
	1996-03-30	LEE	Ontario-North	3
	1996-03-30	LEE	Quebec	7
	1996-03-30	LEE	Manitoba	4
	1996-03-30	GOUNOT	Ontario-South	2
	1996-03-30	GOUNOT	Quebec	18
	1996-03-30	GOUNOT	Manitoba	1
	1996-03-31	LUCCHESSI	Manitoba	1
	1996-03-31	LEE	Ontario-South	14
	1996-03-31	LEE	Ontario-North	3
	1996-03-31	LEE	Quebec	7
	1996-03-31	LEE	Manitoba	3

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
	1996-03-31	GOUNOT	Ontario-South	2
	1996-03-31	GOUNOT	Quebec	1
	1996-04-01	LUCCHESSI	Ontario-South	3
	1996-04-01	LUCCHESSI	Manitoba	1
	1996-04-01	LEE	Ontario-South	8
	1996-04-01	LEE	Ontario-North	?
	1996-04-01	LEE	Quebec	8
	1996-04-01	LEE	Manitoba	9
	1996-04-01	GOUNOT	Ontario-South	3
	1996-04-01	GOUNOT	Ontario-North	1
	1996-04-01	GOUNOT	Quebec	3
	1996-04-01	GOUNOT	Manitoba	7

STAFF

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	SMALLINT NOT NULL	VARCHAR(9)	SMALLINT	CHAR(5)	SMALLINT	DECIMAL(7,2)	DECIMAL(7,2)
Desc:	Staff ID	Name	Department	Job	Years of service	Salary	Commission
Values:	10	Sanders	20	Mgr	7	18357.50	?
	20	Pernal	20	Sales	8	18171.25	612.45
	30	Marenghi	38	Mgr	5	17506.75	?
	40	O'Brien	38	Sales	6	18006.00	846.55
	50	Hanes	15	Mgr	10	20659.80	?
	60	Quigley	38	Sales	?	16808.30	650.25
	70	Rothman	15	Sales	7	16502.83	1152.00
	80	James	20	Clerk	?	13504.60	128.20
	90	Koonitz	42	Sales	6	18001.75	1386.70
	100	Plotz	42	Mgr	7	18352.80	?
	110	Ngan	15	Clerk	5	12508.20	206.60
	120	Naughton	38	Clerk	?	12954.75	180.00
	130	Yamaguchi	42	Clerk	6	10505.90	75.60
	140	Fraye	51	Mgr	6	21150.00	?
	150	Williams	51	Sales	6	19456.50	637.65
	160	Molinare	10	Mgr	7	22959.20	?
	170	Kermisch	15	Clerk	4	12258.50	110.10
	180	Abrahams	38	Clerk	3	12009.75	236.50
	190	Sneider	20	Clerk	8	14252.75	126.50
	200	Scoutten	42	Clerk	?	11508.60	84.20
	210	Lu	10	Mgr	10	20010.00	?
	220	Smith	51	Sales	7	17654.50	992.80
	230	Lundquist	51	Clerk	3	13369.80	189.65
	240	Daniels	10	Mgr	5	19260.25	?
	250	Wheeler	51	Clerk	6	14460.00	513.30
	260	Jones	10	Mgr	12	21234.00	?
	270	Lea	66	Mgr	9	18555.50	?
	280	Wilson	66	Sales	9	18674.50	811.50
	290	Quill	84	Mgr	10	19818.00	?
	300	Davis	84	Sales	5	15454.50	806.10
	310	Graham	66	Sales	13	21000.00	200.30
	320	Gonzales	66	Sales	4	16858.20	844.00

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
	330	Burke	66	Clerk	1	10988.00	55.50
	340	Edwards	84	Sales	7	17844.00	1285.00
	350	Gafney	84	Clerk	5	13030.50	188.00

Sample files with BLOB and CLOB data type

This section shows the data found in the EMP_PHOTO files (pictures of employees) and EMP_RESUME files (resumes of employees).

Quintana photo

Figure 17. Dolores M. Quintana

Quintana resume

Resume: Dolores M. Quintana

Personal Information

Address:
1150 Eglinton Ave Mellonville, Idaho 83725

Phone: (208) 555-9933

Birthdate:
September 15, 1925

Sex: Female

Marital Status:
Married

Height: 5'2"

Weight:
120 lbs.

Department Information

Employee Number:
000130

Dept Number:
C01

Manager:
Sally Kwan

Position:
Analyst

Phone: (208) 555-4578

Hire Date:
1971-07-28

Education

1965 Math and English, B.A. Adelphi University

1960 Dental Technician Florida Institute of Technology

Work History

10/91 - present
Advisory Systems Analyst Producing documentation tools for engineering department.

12/85 - 9/91
Technical Writer, Writer, text programmer, and planner.

1/79 - 11/85
COBOL Payroll Programmer Writing payroll programs for a diesel fuel company.

Interests

- Cooking
- Reading
- Sewing
- Remodeling

Figure 18. Dolores M. Quintana

Nicholls photo

Figure 19. Heather A. Nicholls

Nicholls resume

Resume: Heather A. Nicholls

Personal Information

Address:
844 Don Mills Ave Mellonville, Idaho 83734

Phone: (208) 555-2310

Birthdate:
January 19, 1946

Sex: Female

Marital Status:
Single

Height: 5'8"

Weight:
130 lbs.

Department Information

Employee Number:
000140

Dept Number:
C01

Manager:
Sally Kwan

Position:
Analyst

Phone: (208) 555-1793

Hire Date:
1976-12-15

Education

1972 Computer Engineering, Ph.D. University of Washington

1969 Music and Physics, M.A. Vassar College

Work History

2/83 - present
Architect, OCR Development Designing the architecture of OCR products.

12/76 - 1/83
Text Programmer Optical character recognition (OCR) programming in PL/I.

9/72 - 11/76
Punch Card Quality Analyst Checking punch cards met quality specifications.

Interests

- Model railroading
- Interior decorating
- Embroidery
- Knitting

Figure 20. Heather A. Nicholls

Adamson photo

Figure 21. Bruce Adamson

Adamson resume

Resume: Bruce Adamson

Personal Information

Address:
3600 Steeles Ave Mellonville, Idaho 83757

Phone: (208) 555-4489

Birthdate:
May 17, 1947

Sex: Male

Marital Status:
Married

Height: 6'0"

Weight:
175 lbs.

Department Information

Employee Number:
000150

Dept Number:
D11

Manager:
Irving Stern

Position:
Designer

Phone: (208) 555-4510

Hire Date:
1972-02-12

Education

1971 Environmental Engineering, M.Sc. Johns Hopkins University

1968 American History, B.A. Northwestern University

Work History

8/79 - present
Neural Network Design Developing neural networks for machine intelligence products.

2/72 - 7/79
Robot Vision Development Developing rule-based systems to emulate sight.

9/71 - 1/72
Numerical Integration Specialist Helping bank systems communicate with each other.

Interests

- Racing motorcycles
- Building loudspeakers
- Assembling personal computers
- Sketching

Figure 22. Bruce Adamson

Walker photo

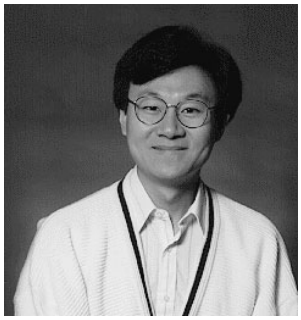


Figure 23. James H. Walker

Walker resume

Resume: James H. Walker

Personal Information

Address:
3500 Steeles Ave Mellonville, Idaho 83757

Phone: (208) 555-7325

Birthdate:
June 25, 1952

Sex: Male

Marital Status:
Single

Height: 5'11"

Weight:
166 lbs.

Department Information

Employee Number:
000190

Dept Number:
D11

Manager:
Irving Stern

Position:
Designer

Phone: (208) 555-2986

Hire Date:
1974-07-26

Education

1974 Computer Studies, B.Sc. University of Massachusetts

1972 Linguistic Anthropology, B.A. University of Toronto

Work History

6/87 - present
Microcode Design Optimizing algorithms for mathematical functions.

4/77 - 5/87
Printer Technical Support Installing and supporting laser printers.

9/74 - 3/77
Maintenance Programming Patching assembly language compiler for mainframes.

Interests

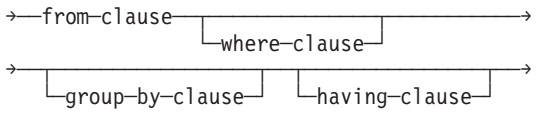
- Wine tasting
- Skiing
- Swimming
- Dancing

Figure 24. James H. Walker

Chapter 21. 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 2011 Core standard terms to DB2 SQL terms.

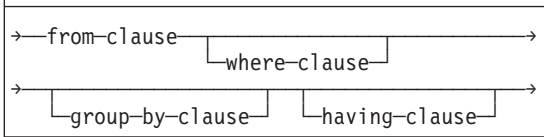
Table 98. ANSI/ISO term to DB2 SQL term cross-reference

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 2011 Core standard terms.

Terminology differences

Table 99. DB2 SQL term to ANSI/ISO term cross-reference

DB2 SQL Term	ANSI/ISO Term
arithmetic expression	value expression
basic predicate	comparison predicate
aggregate function	set function
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
	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 parentheses	query term

Chapter 22. Reserved schema names and reserved words

This appendix 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

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

The DB2 SQL reserved words are:

Table 100. SQL Reserved Words

ACCORDING	CLUSTER	DEFAULT	FIRST
ACCTNG	COLLECT	DEFAULTS	FOR
ACTION	COLLECTION	DEFER	FOREIGN
ACTIVATE	COLLID	DEFINE	FREE
ADD	COLUMN	DEFINITION	FREEPAGE
AFTER	COMMENT	DELETE	FROM
ALIAS	COMMIT	DELETING	FULL
ALL	COMPACT	DENSERANK	FUNCTION
ALLOCATE	COMPRESS	DENSE_RANK	GBPCACHE
ALLOW	CONCAT	DESC	GENERAL
ALTER	CONCURRENT	DESCRIBE	GENERATED
AND	CONDITION	DESCRIPTOR	GET
ANY	CONNECT	DETERMINISTIC	GLOBAL
APPEND	CONNECT_BY_ROOT	DIAGNOSTICS	GO
APPLNAME	CONNECTION	DISABLE	GOTO
ARRAY	CONSTANT	DISALLOW	GRANT
ARRAY_AGG	CONSTRAINT	DISCONNECT	GRAPHIC
ARRAY_EXISTS	CONTAINS	DISTINCT	GROUP
AS	CONTENT	DO	HANDLER
ASC	CONTINUE	DOCUMENT	HASH
ASENSITIVE	COPY	DOUBLE	HASHED_VALUE
ASSOCIATE	COUNT	DROP	HAVING
ASUTIME	COUNT_BIG	DSSIZE	HINT
AT	CREATE	DYNAMIC	HOLD
ATOMIC	CROSS	EACH	HOUR
ATTRIBUTES	CUBE	EDITPROC	HOURS
AUDIT	CURRENT	ELSE	ID
AUTHORIZATION	CURRENT_DATE	ELSEIF	IDENTITY
AUTONOMOUS	CURRENT_LC_CTYPE	ENABLE	IF
AUX	CURRENT_PATH	ENCODING	IGNORE
AUXILIARY	CURRENT_SCHEMA	ENCRYPTION	IMMEDIATE
BEFORE	CURRENT_SERVER	END	IMPLICITLY
BEGIN	CURRENT_TIME	ENDING	IN
BETWEEN	CURRENT_TIMESTAMP	END-EXEC (COBOL only)	INCLUDE
BINARY	CURRENT_TIMEZONE	ENFORCED	INCLUDING
BIND	CURRENT_USER	ERASE	INCLUSIVE
BIT	CURSOR	ESCAPE	INCREMENT
BUFFERPOOL	CURVAL	EVERY	INDEX
BY	CYCLE	EXCEPT	INDEXBP
CACHE	DATA	EXCEPTION	INDICATOR
CALL	DATABASE	EXCLUDING	INDICATORS
CALLED	DATAPARTITIONNAME	EXCLUSIVE	INF
CAPTURE	DATAPARTITIONNUM	EXECUTE	INFINITY
CARDINALITY	DATE	EXISTS	INHERIT
CASCADED	DAY	EXIT	INNER
CASE	DAYS	EXPLAIN	INOUT
CAST	DB2GENERAL	EXTEND	INSENSITIVE
CCSID	DB2GENRL	EXTENDED	INSERT
CHAR	DB2SQL	EXTERNAL	INSERTING
CHARACTER	DBINFO	EXTRACT	INTEGRITY
CHECK	DBPARTITIONNAME	FENCED	INTERSECT
CL	DBPARTITIONNUM	FETCH	INTO
CLIENT	DEACTIVATE	FIELDPROC	IS
CLONE	DEALLOCATE	FILE	ISOBID
CLOSE	DECLARE	FINAL	ISOLATION

Table 101. SQL Reserved Words (continued)

| ITERATE	NEXTVAL	PRIOR	SECOND
JAR	NO	PRIQTY	SECONDS
JAVA	NOCACHE	PRIVILEGES	SECQTY
| JOIN	NOCYCLE	PROCEDURE	SECURED
KEEP	NODENAME	PROGRAM	SECURITY
KEY	NODENUMBER	PROGRAMID	SELECT
LABEL	NOMAXVALUE	PSID	SENSITIVE
LANGUAGE	NOMINVALUE	PUBLIC	SEQUENCE
| LAST	NONE	QUERY	SESSION
LATERAL	NOORDER	QUERYNO	SESSION_USER
LC_CTYPE	NORMALIZED	RANGE	SET
LEAVE	NOT	RANK	SIGNAL
LEFT	NULL	RCDFMT	SIMPLE
LEVEL2	NULLS	READ	SKIP
LIKE	NUMPARTS	READS	SNAN
| LIMIT	NVARCHAR	RECOVERY	SOME
LINKTYPE	OBID	REFERENCES	SOURCE
LOCAL	OF	REFERENCING	SPECIFIC
| LOCALDATE	OFFSET	REFRESH	SQL
LOCALE	OLD	RELEASE	SQLID
LOCALTIME	OLD_TABLE	RENAME	STACKED
LOCALTIMESTAMP	ON	REPEAT	STANDARD
LOCATION	OPEN	RESET	START
LOCATOR	OPTIMIZATION	RESIGNAL	STARTING
LOCATORS	OPTIMIZE	RESTART	STATEMENT
LOCK	OPTION	RESTRICT	STATIC
| LOCKMAX	OR	RESULT	STATMENT
LOCKSIZE	ORDER	RESULT_SET_LOCATOR	STAY
LOG	ORDINALITY	RETURN	STOGROUP
| LOGGED	ORGANIZATION	RETURNS	STORES
LONG	ORGANIZE	REVOKE	STYLE
LOOP	OUT	RID	SUBSTRING
MAINTAINED	OUTER	RIGHT	SUMMARY
| MASK	OVER	ROLE	SYNONYM
| MATCHED	OVERRIDING	ROLLBACK	SYSDATESYSTEM
MATERIALIZED	PACKAGE	ROLLUP	SYSFUN
MAXVALUE	PADDED	ROUND_CEILING	SYSIBM
MERGE	PAGE	ROUND_DOWN	SYSPROC
MICROSECOND	PAGESIZE	ROUND_FLOOR	SYSTEM
MICROSECONDS	PARAMETER	ROUND_HALF_DOWN	SYSTEM_USER
| MINPCTUSED	PART	ROUND_HALF_EVEN	SYSTIMESTAMP
MINUTE	PARTITION	ROUND_HALF_UP	TABLE
MINUTES	PARTITIONED	ROUND_UP	TABLESPACE
MINVALUE	PARTITIONING	ROUTINE	TABLESPACES
MIXED	PARTITIONS	ROW	THEN
MODE	PASSWORD	ROWNUMBER	THREADSAFE
MODIFIES	PATH	ROWS	TIME
MONTH	PCTFREE	ROWSET	TIMESTAMP
| MONTHS	PERIOD	ROW_NUMBER	TO
| NAMESPACE	PERMISSION	RRN	TRANSACTION
| NAN	PIECESIZE	RUN	TRANSFER
NATIONAL	PLAN	SAVEPOINT	TRIGGER
NCHAR	POSITION	SBCS	TRIM
NCLOB	PRECISION	SCHEMA	TRIM_ARRAY
NEW	PREPARE	SCRATCHPAD	TRUNCATE
NEW_TABLE	PREVVAL	SCROLL	TYPE
| NEXT	PRIMARY	SEARCH	UNDO

Table 102. SQL Reserved Words (continued)

UNION	VALUES	WLM	XMLNAMESPACES
UNIQUE	VARIABLE	WRAPPED	XMLPARSE
UNIT	VARIANT	WRITE	XMLPI
UNNEST	VCAT	WRKSTNNAME	XMLROW
UNTIL	VERSION	XMLAGG	XMLSERIALIZE
UPDATE	VERSIONING	XMLATTRIBUTES	XMLTABLE
UPDATING	VIEW	XMLCAST	XMLTEXT
URI	VOLATILE	XMLCOMMENT	XMLVALIDATE
USAGE	VOLUMESWAIT	XMLCONCAT	XSLTRANSFORM
USE	WHEN	XMLDOCUMENT	XSROBJECT
USER	WHENEVER	XMLELEMENT	YEAR
USERID	WHERE	XMLEXISTS	YEARS
USING	WHILE	XMLFOREST	YES
VALIDPROC	WITH	XMLGROUP	ZONE
VALUE	WITHOUT		

The ISO/ANSI SQL2011 reserved words that are not in the list of DB2 SQL reserved words are:

Table 103. Additional ISO/ANS Reserved Words

ABS	ELEMENT	NULLIF	SQLSTATE
ARE	EXEC	NUMERIC	SQLWARNING
ASYMMETRIC	EXP	OCTET_LENGTH	SQRT
AVG	FALSE	ONLY	STDDEV_POP
BIGINT	FILTER	OVERLAPS	STDDEV_SAMP
BLOB	FLOAT	OVERLAY	SUBMULTISET
BOOLEAN	FLOOR	PERCENT_RANK	SUM
BOTH	FUSION	PERCENTILE_CONT	SYMMETRIC
CEIL	GROUPING	PERCENTILE_DISC	TABLESAMPLE
CEILING	INT	POWER	TIMEZONE_HOUR
CHAR_LENGTH	INTEGER	REAL	TIMEZONE_MINUTE
CHARACTER_LENGTH	INTERSECTION	RECURSIVE	TRAILING
CLOB	INTERVAL	REF	TRANSLATE
COALESCE	LARGE	REGR_AVGX	TRANSLATION
COLLATE	LEADING	REGR_AVGY	TREAT
CONVERT	LN	REGR_COUNT	TRUE
CORR	LOWER	REGR_INTERCEPT	UESCAPE
CORRESPONDING	MATCH	REGR_R2	UNKNOWN
COVAR_POP	MAX	REGR_SLOPE	UPPER
COVAR_SAMP	MEMBER	REGR_SXX	VAR_POP
CUME_DIST	METHOD	REGR_SXY	VAR_SAMP
CURRENT_DEFAULT_TRANSFORM_GROUP	MIN	REGR_SYY	VARCHAR
CURRENT_ROLE	MOD	SCOPE	VARYING
CURRENT_TRANSFORM_GROUP_FOR_TYPE	MODULE	SIMILAR	WIDTH_BUCKET
DEC	MULTISET	SMALLINT	WINDOW
DECIMAL	NATURAL	SPECIFICTYPE	WITHIN
DEREF	NORMALIZE	SQLEXCEPTION	

Chapter 23. 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 give 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:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

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 Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

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 information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, 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. 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 IBM's application programming interfaces.

Programming interface information

This *SQL Reference for Cross-Platform Development* documents intended Programming Interfaces that allow the customer to write programs to obtain the services of DB2 products.

Trademarks

Company, product, or service names identified in the DB2 SQL Reference for Cross-Platform Development Version 5.0 may be trademarks or service marks of International Business Machines Corporation or other companies. Information about the trademarks of IBM Corporation in the United States, other countries, or both is located at http://www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

F

G

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y