IBM® DB2 Universal Database™

SQL Reference Volume 2

Version 8

SC09-4845-00

IBM® DB2 Universal Database™

SQL Reference Volume 2

Version 8

SC09-4845-00

Before using this information and the product it supports, be sure to read the general information under *Notices*.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this book

The SQL Reference in its two volumes defines the SQL language used by DB2 Universal Database Version 8, and includes:

- Information about relational database concepts, language elements, functions, and the forms of queries (Volume 1).
- Information about the syntax and semantics of SQL statements (Volume 2).

Who should use this book

This book is intended for anyone who wants to use the Structured Query Language (SQL) to access a database. It is primarily for programmers and database administrators, but it can also be used by those who access databases through the command line processor (CLP).

This book is a reference rather than a tutorial. It assumes that you will be writing application programs and therefore presents the full functions of the database manager.

How this book is structured

This book contains information about the following major topics:

- Chapter 1, "Statements" on page 1 contains syntax diagrams, semantic descriptions, rules, and examples of all SQL statements.
- Chapter 2, "SQL control statements" on page 755 contains syntax diagrams, semantic descriptions, rules, and examples of SQL procedure statements.

A brief overview of Volume 1

The first volume of the SQL Reference contains information about relational database concepts, language elements, functions, and the forms of queries. The specific chapters and appendixes in that volume are briefly described here:

- "Concepts" discusses the basic concepts of relational databases and SQL.
- "Language elements" describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- "Functions" contains syntax diagrams, semantic descriptions, rules, and usage examples of SQL column and scalar functions.
- "Queries" describes the various forms of a query.
- "SQL limits" lists the SQL limitations.
- "SQL communications area (SQLCA)" describes the SQLCA structure.

A brief overview of Volume 1

- "SQL descriptor area (SQLDA)" describes the SQLDA structure.
- "Catalog views" describes the database catalog views.
- "Federated systems" describes options and type mappings for federated systems.
- "Sample database tables" describes the sample tables used in examples.
- "Reserved schema names and reserved words" contains the reserved schema names and the reserved words for the IBM SQL and ISO/ANS SQL99 standards.
- "Comparison of isolation levels" contains a summary of the isolation levels.
- "Interaction of triggers and constraints" discusses the interaction of triggers and referential constraints.
- "Explain tables" describes the Explain tables.
- "Explain register values" describes the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values with each other and with the PREP and BIND commands.
- "Recursion example: bill of materials" contains an example of a recursive query.
- "Exception tables" contains information about user-created tables that are used with the SET INTEGRITY statement.
- "SQL statements allowed in routines" lists the SQL statements that are allowed to execute in routines with different SQL data access contexts.
- "CALL" describes the CALL statement that can be invoked from a compiled statement.
- "Japanese and traditional-Chinese EUC considerations" lists considerations when using extended UNIX code (EUC) character sets.
- "BNF specifications for DATALINKs" contains the Backus-Naur form (BNF) specifications for DATALINKs.

How to read the syntax diagrams

Throughout this book, syntax is described using the structure defined as follows:

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

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

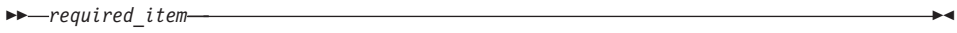
The —► symbol indicates that the syntax is continued on the next line.

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

The ⟶◂ symbol indicates the end of a syntax diagram.

Syntax fragments 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 optional item appears above the main path, that item has no effect on execution, and is used only for readability.

If you can choose from two or more items, they appear in a stack.

If you *must* choose one of the items, 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.

An arrow returning to the left, above the main line, indicates an item that can be repeated. In this case, repeated items must be separated by one or more blanks.

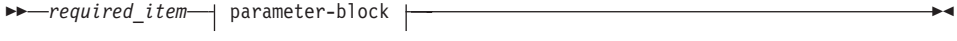
If the repeat arrow contains a comma, you must separate repeated items with a comma.

A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

Keywords appear in uppercase (for example, `FROM`). They must be spelled exactly as shown. Variables appear in lowercase (for example, `column-name`). They represent user-supplied names or values in the syntax.

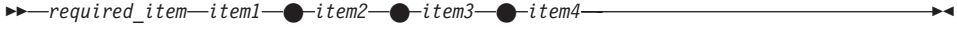
If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

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:

Adjacent segments occurring between "large bullets" (●) may be specified in any sequence.



The above diagram shows that item2 and item3 may be specified in either order. Both of the following are valid:

```
required_item item1 item2 item3 item4
required_item item1 item3 item2 item4
```

Common syntax elements

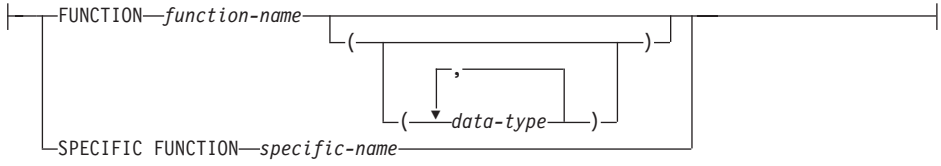
The following sections describe a number of syntax fragments that are used in syntax diagrams. The fragments are referenced as follows:

Function designator

A function designator uniquely identifies a single function. Function designators typically appear in DDL statements for functions (such as DROP or ALTER).

Syntax:

function-designator:



Description:

FUNCTION *function-name*
Identifies a particular function, and is valid only if there is exactly one function instance with the name *function-name* in the schema. The identified function can have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one instance of the function in the named or implied schema, an error (SQLSTATE 42725) is raised.

FUNCTION *function-name (data-type,...)*
Provides the function signature, which uniquely identifies the function. The function resolution algorithm is not used.

function-name
Specifies the name of the function. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an

unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE FUNCTION statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific function instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

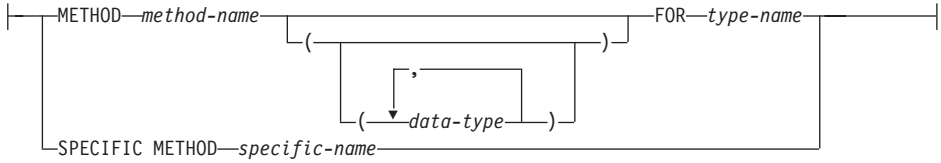
SPECIFIC FUNCTION *specific-name*
Identifies a particular user-defined function, using the name that is specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

Method designator

A method designator uniquely identifies a single method. Method designators typically appear in DDL statements for methods (such as DROP or ALTER).

Syntax:

method-designator:



Description:

METHOD *method-name*
Identifies a particular method, and is valid only if there is exactly one method instance with the name *method-name* for the type *type-name*. The identified method can have any number of parameters defined for it. If no method by this name exists for the type, an error (SQLSTATE 42704) is raised. If there is more than one instance of the method for the type, an error (SQLSTATE 42725) is raised.

METHOD *method-name (data-type,...)*
Provides the method signature, which uniquely identifies the method. The method resolution algorithm is not used.

method-name
Specifies the name of the method for the type *type-name*.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE TYPE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific method instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE TYPE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no method with the specified signature exists for the type in the named or implied schema, an error (SQLSTATE 42883) is raised.

FOR *type-name*
Names the type with which the specified method is to be associated. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

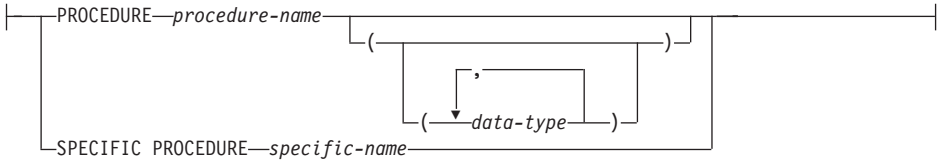
SPECIFIC METHOD *specific-name*
Identifies a particular method, using the name that is specified or defaulted to at method creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific method instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

Procedure designator

A procedure designator uniquely identifies a single stored procedure. Procedure designators typically appear in DDL statements for procedures (such as DROP or ALTER).

Syntax:

procedure-designator:



Description:

PROCEDURE *procedure-name*
Identifies a particular procedure, and is valid only if there is exactly one procedure instance with the name *procedure-name* in the schema. The identified procedure can have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no procedure by this name exists in the

named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one instance of the procedure in the named or implied schema, an error (SQLSTATE 42725) is raised.

PROCEDURE *procedure-name (data-type,...)*
Provides the procedure signature, which uniquely identifies the procedure. The procedure resolution algorithm is not used.

procedure-name
Specifies the name of the procedure. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE PROCEDURE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific procedure instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE PROCEDURE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC PROCEDURE *specific-name*
Identifies a particular procedure, using the name that is specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified

object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

Conventions used in this manual

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

Error conditions

An error condition is indicated within the text of the manual by listing the SQLSTATE associated with the error in parentheses. For example:

```
A duplicate signature raises an SQL error (SQLSTATE 42723).
```

Highlighting conventions

The following conventions are used in this book.

Bold	Indicates commands, keywords, and other items whose names are predefined by the system.
Italics	Indicates one of the following: • Names or values (variables) that must be supplied by the user. • General emphasis. • The introduction of a new term. • A reference to another source of information.
`Monospace`	Indicates one of the following: • Files and directories. • Information that you are instructed to type at a command prompt or in a window. • Examples of specific data values. • Examples of text similar to what may be displayed by the system. • Examples of system messages.

Related documentation

The following publications may prove useful in preparing applications:

- *Administration Guide*
  - Contains information required to design, implement, and maintain a database to be accessed either locally or in a client/server environment.
- *Application Development Guide*
  - Discusses the application development process and how to code, compile, and execute application programs that use embedded SQL and APIs to access the database.

- *DB2 Universal Database for iSeries SQL Reference*
  - This book defines Structured Query Language (SQL) as supported by DB2 Query Manager and SQL Development Kit on iSeries (AS/400). It contains reference information for the tasks of system administration, database administration, application programming, and operation. This manual includes syntax, usage notes, keywords, and examples for each of the SQL statements used on iSeries (AS/400) systems running DB2.
- *DB2 Universal Database for z/OS and OS/390 SQL Reference*
  - This book defines Structured Query Language (SQL) used in DB2 for z/OS (OS/390). It provides query forms, SQL statements, SQL procedure statements, DB2 limits, SQLCA, SQLDA, catalog tables, and SQL reserved words for z/OS (OS/390) systems running DB2.
- *DB2 Spatial Extender User's Guide and Reference*
  - This book discusses how to write applications to create and use a geographic information system (GIS). Creating and using a GIS involves supplying a database with resources and then querying the data to obtain information such as locations, distances, and distributions within areas.
- *IBM SQL Reference*
  - This book contains all the common elements of SQL that span IBM's database products. It provides limits and rules that assist in preparing portable programs using IBM databases. This manual provides a list of SQL extensions and incompatibilities among the following standards and products: SQL92E, XPG4-SQL, IBM-SQL and the IBM relational database products.
- *American National Standard X3.135-1992, Database Language SQL*
  - Contains the ANSI standard definition of SQL.
- *ISO/IEC 9075:1992, Database Language SQL*
  - Contains the 1992 ISO standard definition of SQL.
- *ISO/IEC 9075-2:1999, Database Language SQL -- Part 2: Foundation (SQL/Foundation)*
  - Contains a large portion of the 1999 ISO standard definition of SQL.
- *ISO/IEC 9075-4:1999, Database Language SQL -- Part 4: Persistent Stored Modules (SQL/PSM)*
  - Contains the 1999 ISO standard definition for SQL procedure control statements.
- *ISO/IEC 9075-5:1999, Database Language SQL -- Part 4: Host Language Bindings (SQL/Bindings)*
  - Contains the 1999 ISO standard definition for host language bindings and dynamic SQL.

Related documentation

Chapter 1. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements.

Table 1. SQL Statements

SQL Statement	Function	Page
ALTER BUFFERPOOL	Changes the definition of a buffer pool.	12
ALTER DATABASE PARTITION GROUP	Changes the definition of a database partition group.	15
ALTER FUNCTION	Modifies an existing function by changing the properties of the function.	19
ALTER METHOD	Modifies an existing method by changing the method body associated with the method.	22
ALTER NICKNAME	Changes the definition of a nickname.	24
ALTER PROCEDURE	Modifies an existing procedure by changing the properties of the procedure.	28
ALTER SEQUENCE	Changes the definition of a sequence.	32
ALTER SERVER	Changes the definition of a data source in a federated system.	37
ALTER TABLE	Changes the definition of a table.	41
ALTER TABLESPACE	Changes the definition of a table space.	75
ALTER TYPE (Structured)	Changes the definition of a structured type.	83
ALTER USER MAPPING	Changes the definition of a user authorization mapping.	92
ALTER VIEW	Changes the definition of a view by altering a reference type column to add a scope.	95
ALTER WRAPPER	Updates the options that, along with a wrapper module, are used to access data sources of a specific type.	97
BEGIN DECLARE SECTION	Marks the beginning of a host variable declaration section.	98
CALL	Calls a stored procedure.	101
CLOSE	Closes a cursor.	107
COMMENT	Replaces or adds a comment to the description of an object.	109
COMMIT	Terminates a unit of work and commits the database changes made by that unit of work.	120
Compound SQL (Dynamic)	Combines one or more other SQL statements into an dynamic block.	123

Statements

Table 1. SQL Statements (continued)

SQL Statement	Function	Page
Compound SQL (Embedded)	Combines one or more other SQL statements into an executable block.	129
CONNECT (Type 1)	Connects to an application server according to the rules for remote unit of work.	134
CONNECT (Type 2)	Connects to an application server according to the rules for application-directed distributed unit of work.	142
CREATE ALIAS	Defines an alias for a table, view, or another alias.	151
CREATE BUFFERPOOL	Creates a new buffer pool.	154
CREATE DATABASE PARTITION GROUP	Defines a database partition group.	158
CREATE DISTINCT TYPE	Defines a distinct data type.	161
CREATE EVENT MONITOR	Specifies events in the database to monitor.	168
CREATE FUNCTION	Registers a user-defined function.	188
CREATE FUNCTION (External Scalar)	Registers a user-defined external scalar function.	190
CREATE FUNCTION (External Table)	Registers a user-defined external table function.	217
CREATE FUNCTION (OLE DB External Table)	Registers a user-defined OLE DB external table function.	235
CREATE FUNCTION (Sourced or Template)	Registers a user-defined sourced function.	243
CREATE FUNCTION (SQL Scalar, Table or Row)	Registers and defines a user-defined SQL function.	254
CREATE FUNCTION MAPPING	Defines a function mapping.	263
CREATE INDEX	Defines an index on a table.	268
CREATE INDEX EXTENSION	Defines an extension object for use with indexes on tables with structured or distinct type columns.	277
CREATE METHOD	Associates a method body with a previously defined method specification.	285
CREATE NICKNAME	Defines a nickname.	291
CREATE PROCEDURE	Registers a stored procedure.	296
CREATE PROCEDURE (External)	Registers an external stored procedure.	297

Table 1. SQL Statements (continued)

SQL Statement	Function	Page
CREATE PROCEDURE (SQL)	Registers an SQL stored procedure.	311
CREATE SCHEMA	Defines a schema.	318
CREATE SEQUENCE	Defines a sequence.	322
CREATE SERVER	Defines a data source to a federated database.	328
CREATE TABLE	Defines a table.	332
CREATE TABLESPACE	Defines a table space.	395
CREATE TRANSFORM	Defines transformation functions.	405
CREATE TRIGGER	Defines a trigger.	414
CREATE TYPE (Structured)	Defines a structured data type.	427
CREATE TYPE MAPPING	Defines a mapping between data types.	456
CREATE USER MAPPING	Defines a mapping between user authorizations.	461
CREATE VIEW	Defines a view of one or more table, view or nickname.	463
CREATE WRAPPER	Registers a wrapper.	479
DECLARE CURSOR	Defines an SQL cursor.	482
DECLARE GLOBAL TEMPORARY TABLE	Defines the Global Temporary Table.	488
DELETE	Deletes one or more rows from a table.	497
DESCRIBE	Describes the result columns of a prepared SELECT statement.	504
DISCONNECT	Terminates one or more connections when there is no active unit of work.	509
DROP	Deletes objects in the database.	512
END DECLARE SECTION	Marks the end of a host variable declaration section.	542
EXECUTE	Executes a prepared SQL statement.	544
EXECUTE IMMEDIATE	Prepares and executes an SQL statement.	552
EXPLAIN	Captures information about the chosen access plan.	556
FETCH	Assigns values of a row to host variables.	561
FLUSH EVENT MONITOR	Writes out the active internal buffer of an event monitor.	565
FLUSH PACKAGE CACHE	Removes all cached dynamic SQL statements currently in the package cache.	566

Table 1. SQL Statements (continued)

SQL Statement	Function	Page
FREE LOCATOR	Removes the association between a locator variable and its value.	567
GRANT (Database Authorities)	Grants authorities on the entire database.	569
GRANT (Index Privileges)	Grants the CONTROL privilege on indexes in the database.	573
GRANT (Package Privileges)	Grants privileges on packages in the database.	575
GRANT (Routine Privileges)	Grants privileges on a routine (function, method, or procedure).	579
GRANT (Schema Privileges)	Grants privileges on a schema.	583
GRANT (Sequence Privileges)	Grants privileges on a sequence.	586
GRANT (Server Privileges)	Grants privileges to query a specific data source.	588
GRANT (Table, View, or Nickname Privileges)	Grants privileges on tables, views and nicknames.	590
GRANT (Table Space Privileges)	Grants privileges on a tablespace.	598
INCLUDE	Inserts code or declarations into a source program.	601
INSERT	Inserts one or more rows into a table.	603
LOCK TABLE	Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table.	613
OPEN	Prepares a cursor that will be used to retrieve values when the FETCH statement is issued.	615
PREPARE	Prepares an SQL statement (with optional parameters) for execution.	620
REFRESH TABLE	Refreshes the data in a materialized query table.	632
RELEASE (Connection)	Places one or more connections in the release-pending state.	634
RELEASE SAVEPOINT	Releases a savepoint within a transaction.	636
RENAME	Renames an existing table.	637
RENAME TABLESPACE	Renames an existing tablespace.	640
REVOKE (Database Authorities)	Revokes authorities from the entire database.	642
REVOKE (Index Privileges)	Revokes the CONTROL privilege on given indexes.	647

Table 1. SQL Statements (continued)

SQL Statement	Function	Page
REVOKE (Package Privileges)	Revokes privileges from given packages in the database.	650
REVOKE (Routine Privileges)	Revokes privileges on a routine (function, method, or procedure).	653
REVOKE (Schema Privileges)	Revokes privileges on a schema.	657
REVOKE (Server Privileges)	Revokes privileges to query a specific data source.	660
REVOKE (Table, View, or Nickname Privileges)	Revokes privileges from given tables, views or nicknames.	662
REVOKE (Table Space Privileges)	Revokes the USE privilege on a given table space.	668
ROLLBACK	Terminates a unit of work and backs out the database changes made by that unit of work.	671
SAVEPOINT	Sets a savepoint within a transaction.	674
SELECT INTO	Specifies a result table of no more than one row and assigns the values to host variables.	677
SET CONNECTION	Changes the state of a connection from dormant to current, making the specified location the current server.	680
SET CURRENT DEFAULT TRANSFORM GROUP	Changes the value of the CURRENT DEFAULT TRANSFORM GROUP special register.	683
SET CURRENT DEGREE	Changes the value of the CURRENT DEGREE special register.	685
SET CURRENT EXPLAIN MODE	Changes the value of the CURRENT EXPLAIN MODE special register.	687
SET CURRENT EXPLAIN SNAPSHOT	Changes the value of the CURRENT EXPLAIN SNAPSHOT special register.	689
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Changes the value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register.	691
SET CURRENT PACKAGESET	Sets the schema name for package selection.	693
SET CURRENT QUERY OPTIMIZATION	Changes the value of the CURRENT QUERY OPTIMIZATION special register.	695
SET CURRENT REFRESH AGE	Changes the value of the CURRENT REFRESH AGE special register.	698
SET ENCRYPTION PASSWORD	Sets the password for encryption.	700

Table 1. SQL Statements (continued)

SQL Statement	Function	Page
SET EVENT MONITOR STATE	Activates or deactivates an event monitor.	702
SET INTEGRITY	Sets the check pending state and checks data for constraint violations.	704
SET PASSTHRU	Opens a session for submitting data source native SQL directly to the data source.	724
SET PATH	Changes the value of the CURRENT PATH special register.	726
SET SCHEMA	Changes the value of the CURRENT SCHEMA special register.	729
SET SERVER OPTION	Sets server option settings.	731
SET Variable	Assigns values to NEW transition variables.	733
UPDATE	Updates the values of one or more columns in one or more rows of a table.	738
VALUES INTO	Specifies a result table of no more than one row and assigns the values to host variables.	751
WHENEVER	Defines actions to be taken on the basis of SQL return codes.	753

How SQL statements are invoked

SQL statements are classified as executable or non-executable.

An *executable statement* can be invoked in four ways. It can be:

- Embedded in an application program
- Embedded in an SQL procedure.
- Prepared and executed dynamically
- Issued interactively

Depending on the statement, some or all of these methods can be used. (Statements embedded in REXX are prepared and executed dynamically.)

A *non-executable statement* can only be embedded in an application program.

Another SQL statement construct is the select-statement. A *select-statement* can be invoked in three ways. It can be:

- Included in DECLARE CURSOR, and executed implicitly by OPEN, FETCH and CLOSE (static invocation)
- Prepared dynamically, referenced in DECLARE CURSOR, and executed implicitly by OPEN, FETCH and CLOSE (dynamic invocation)
- Issued interactively

Embedding a statement in an application program

SQL statements can be included in a source program that will be submitted to a 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 the keywords EXEC SQL.

Executable statements

An executable statement embedded in an application program is executed every time a statement of the host language would be executed if it were 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 host variables. A host variable referenced in this way can be used in two ways. It can be used:

- As input (the current value of the host 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 host variables in expressions and predicates are effectively replaced by current values of the variables; that is, the variables are used as input.

All executable statements should be followed by a test of the SQL return code. Alternatively, the WHENEVER statement (which is itself non-executable) can be used to change the flow of control immediately after the execution of an embedded statement.

All objects referenced in data manipulation language (DML) statements must exist when the statements are bound to a database.

Non-executable statements

An embedded non-executable statement is processed only by the precompiler. The precompiler reports any errors encountered in the statement. The statement is *never* processed during program execution; therefore, such statements should not be followed by a test of the SQL return code.

Embedding a statement in an SQL procedure

Statements can be included in the SQL-procedure-body portion of the CREATE PROCEDURE statement. Such statements are said to be embedded in the SQL procedure. Whenever an SQL statement description refers to a *host-variable*, an *SQL-variable* can be used if the statement is embedded in an SQL procedure.

Dynamic preparation and execution

An application program can dynamically build an SQL statement in the form of a character string placed in a host variable. In general, the statement is built from some data available to the program (for example, input from a workstation). The statement (not a select-statement) constructed can be prepared for execution by means of the (embedded) PREPARE statement, and executed by means of the (embedded) EXECUTE statement. Alternatively, an (embedded) EXECUTE IMMEDIATE statement can be used to prepare and execute the statement in one step.

A statement that is going to be dynamically prepared must not contain references to host variables. It can instead contain parameter markers. (For rules concerning parameter markers, see "PREPARE".) When the prepared statement is executed, the parameter markers are effectively replaced by current values of the host variables specified in the EXECUTE statement. Once prepared, a statement can be executed several times with different values for the host variables. Parameter markers are not allowed in the EXECUTE IMMEDIATE statement.

Successful or unsuccessful execution of the statement is indicated by the setting of an SQL return code in the SQLCA after the EXECUTE (or EXECUTE

IMMEDIATE) statement completes. The SQL return code should be checked, as described above. For more information, see "SQL return codes" on page 10.

Static invocation of a select-statement

A select-statement can be included as a part of the (non-executable) DECLARE CURSOR statement. Such a statement is executed every time the cursor is opened by means of the (embedded) OPEN statement. 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 host variables. These references are effectively replaced by the values that the variables have when the OPEN statement executes.

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 host 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) PREPARE statement, and referenced by a (non-executable) DECLARE CURSOR statement. The statement is then executed every time the cursor is opened by means of the (embedded) OPEN statement. 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 host variables. It can contain parameter markers instead. The parameter markers are effectively replaced by the values of the host variables specified in the OPEN statement.

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. Such a statement must be an executable statement that does not contain parameter markers or references to host variables, because these make sense only in the context of an application program.

SQL use with other host systems

SQL statement syntax exhibits minor variations among different types of host systems (DB2 for z/OS, DB2 for iSeries, DB2 Universal Database). Regardless of whether the SQL statements in an application are static or dynamic, it is important — if the application is meant to access different database host systems — to ensure that the SQL statements and precompile/bind options are supported on the database systems that the application will access.

Further information about SQL statements used in other host systems can be found in the *DB2 Universal Database for iSeries SQL Reference* and the *DB2 Universal Database for OS/390 and z/OS SQL Reference*.

SQL return codes

An application program containing executable SQL statements can use either SQLCODE or SQLSTATE values to handle return codes from SQL statements. There are two ways in which an application can get access to these values.

- Include a structure named SQLCA. The SQLCA includes an integer variable named SQLCODE and a character string variable named SQLSTATE. In REXX, an SQLCA is provided automatically. In other languages, an SQLCA can be obtained by using the INCLUDE SQLCA statement.
- If LANGLEVEL SQL92E is specified as a precompile option, a variable named SQLCODE or SQLSTATE can be declared in the SQL declare section of the program. If neither of these variables is declared in the SQL declare section, it is assumed that a variable named SQLCODE is declared elsewhere in the program. With LANGLEVEL SQL92E, the program should not have an INCLUDE SQLCA statement.

SQLCODE

An SQLCODE is set by the database manager after each SQL statement executes. All database managers conform to the ISO/ANSI SQL standard, 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, but one or more warning indicators were set.
- If SQLCODE < 0, execution was not successful.

The meaning of SQLCODE values other than 0 and 100 is product-specific.

SQLSTATE

An SQLSTATE is set by the database manager after each SQL statement executes. Application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE. SQLSTATE provides common codes for common error conditions. Application programs can test for specific errors or classes of errors. The coding scheme is the same for all IBM database managers, and is based on the ISO/ANSI SQL92 standard.

SQL comments

Static SQL statements can include host language or SQL comments. SQL comments are introduced by two hyphens.

The following rules apply to the use of SQL comments:

- The two hyphens must be on the same line, not separated by a space.
- Comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Comments are terminated by the end of the line.
- Comments are not allowed within statements that are dynamically prepared (using PREPARE or EXECUTE IMMEDIATE).
- In COBOL, the hyphens must be preceded by a space.

Example: This example shows how to include comments in an SQL statement within a C program:

```
EXEC SQL
  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;
```

Related reference:

- "Select-statement" in the *SQL Reference, Volume 1*
- "EXECUTE" on page 544
- "OPEN" on page 615
- "PREPARE" on page 620
- "SQLCA (SQL communications area)" in the *SQL Reference, Volume 1*
- "SQL procedure statement" on page 757

ALTER BUFFERPOOL

The ALTER BUFFERPOOL statement is used to do the following:

- modify the size of the buffer pool on all partitions or on a single partition
- turn on or off the use of extended storage
- add this buffer pool definition to a new database partition group
- modify the block area of the buffer pool for block-based I/O.

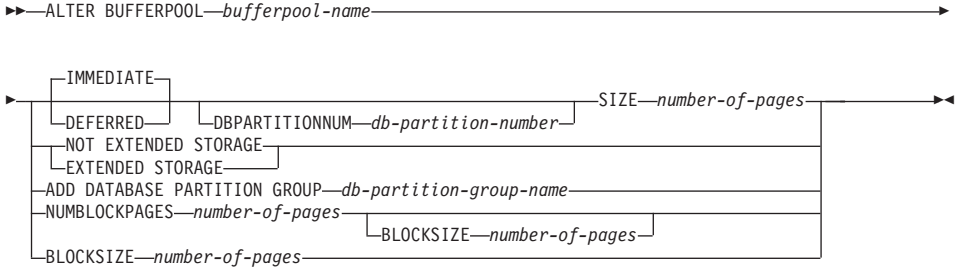
Invocation:

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement must have SYSCTRL or SYSADM authority.

Syntax:



Description:

bufferpool-name
Names the buffer pool. This is a one-part name. It is an SQL identifier (either ordinary or delimited). It must be a buffer pool described in the catalog.

DBPARTITIONNUM *db-partition-number*
Specifies the partition on which size of the buffer pool is modified. The partition must be in one of the database partition groups for the buffer pool (SQLSTATE 42729). If this clause is not specified, then the size of the buffer pool is modified on all partitions on which the buffer pool exists that used the default size for the buffer pool (did not have a size specified in the *except-on-db-partitions-clause* of the CREATE BUFFERPOOL statement).

SIZE *number-of-pages*
The size of the buffer pool specified as the number of pages.

IMMEDIATE
The bufferpool size will be changed immediately. If there is not enough reserved space in the database shared memory to allocate new space (SQLSTATE 01657), the statement is executed as DEFERRED.

DEFERRED
The bufferpool size will be changed when the database is reactivated (all applications need to be disconnected from the database). Reserved memory space is not needed; DB2 will allocate the required memory from the system at activation time.

NOT EXTENDED STORAGE
Even if extended storage is enabled, pages that are being evicted from this buffer pool are not cached in extended storage.

EXTENDED STORAGE
If extended storage is enabled, it can be used as a secondary cache for pages that are evicted from the buffer pool. (Extended storage is enabled by setting the database configuration parameters NUM_ESTORE_SEGS and ESTORE_SEG_SIZE to non-zero values.)

ADD DATABASE PARTITION GROUP *db-partition-group-name*
Adds this database partition group to the list of database partition groups to which the buffer pool definition is applicable. For any partition in the database partition group that does not already have the bufferpool defined, the bufferpool is created on the partition using the default size specified for the bufferpool. Table spaces in *db-partition-group-name* may specify this buffer pool. The database partition group must currently exist in the database (SQLSTATE 42704).

NUMBLOCKPAGES *number-of-pages*
Specifies the number of pages that should exist in the block-based area. The number of pages must not be greater than 98 percent of the number of pages for the buffer pool (SQLSTATE 54052). Specifying the value 0 disables block I/O. The actual value of NUMBLOCKPAGES used will be a multiple of BLOCKSIZE.

BLOCKSIZE *number-of-pages*
Specifies the number of pages in a block. The block size must be a value between 2 and 256 (SQLSTATE 54053). The default value is 32.

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NODE can be specified in place of DBPARTITIONNUM

  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP

- Only the buffer pool size can be changed dynamically (immediately). All other changes are deferred, and will only come into effect after the database is reactivated.
- If the statement is executed as deferred, the following is true: Although the buffer pool definition is transactional and the changes to the buffer pool definition will be reflected in the catalog tables on commit, no changes to the actual buffer pool will take effect until the next time the database is started. The current attributes of the buffer pool will exist until then, and there will not be any impact to the buffer pool in the interim. Tables created in table spaces of new database partition groups will use the default buffer pool. The statement is IMMEDIATE by default when that keyword applies.
- There should be enough real memory on the machine for the total of all the buffer pools, as well as for the rest of the database manager and application requirements.
- A buffer pool that is currently using extended storage cannot be altered to use block-based input/output (I/O). A buffer pool cannot be altered to use both extended storage and block-based I/O simultaneously.

ALTER DATABASE PARTITION GROUP

The ALTER DATABASE PARTITION GROUP statement is used to:

- add one or more partitions to a database partition group
- drop one or more partitions from a database partition group.

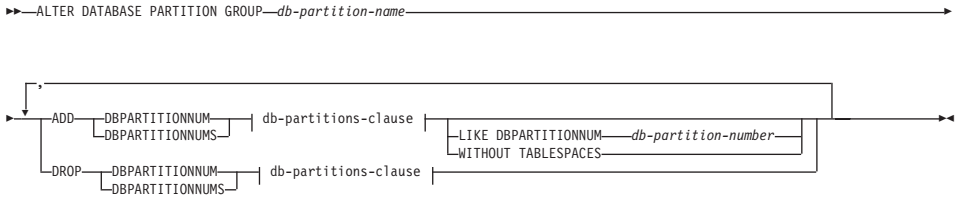
Invocation:

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

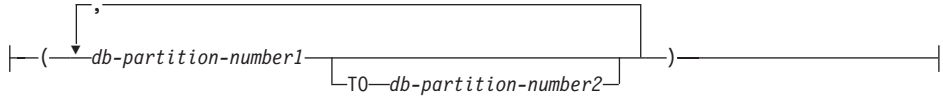
Authorization:

The authorization ID of the statement must have SYSCTRL or SYSADM authority.

Syntax:



db-partitions-clause:



Description:

db-partition-name
Names the database partition group. This is a one-part name. It is an SQL identifier (either ordinary or delimited). It must be a database partition group described in the catalog. IBMCATGROUP and IBMTEMPGROUP cannot be specified (SQLSTATE 42832).

ADD DBPARTITIONNUM
Specifies the specific partition or partitions to add to the database partition group. DBPARTITIONNUMS is a synonym for DBPARTITIONNUM. Any specified partition must not already be defined in the database partition group (SQLSTATE 42728).

DROP DBPARTITIONNUM
Specifies the specific partition or partitions to drop from the database partition group. DBPARTITIONNUMS is a synonym for DBPARTITIONNUM. Any specified partition must already be defined in the database partition group (SQLSTATE 42729).

db-partitions-clause
Specifies the partition or partitions to be added or dropped.

db-partition-number1
Specify a specific partition number.

TO *db-partition-number2*
Specify a range of partition numbers. The value of *db-partition-number2* must be greater than or equal to the value of *db-partition-number1* (SQLSTATE 428A9).

LIKE DBPARTITIONNUM *db-partition-number*
Specifies that the containers for the existing table spaces in the database partition group will be the same as the containers on the specified *db-partition-number*. The partition specified must be a partition that existed in the database partition group prior to this statement and is not included in a DROP DBPARTITIONNUM clause of the same statement.

WITHOUT TABLESPACES
Specifies that the default table spaces are not created on the newly added partition or partitions. The ALTER TABLESPACE statement using the FOR DBPARTITIONNUM clause must be used to define containers for use with the table spaces that are defined on this database partition group. If this option is not specified, the default containers are specified on newly added partitions for each table space defined on the database partition group.

Rules:

- Each partition specified by number must be defined in the `db2nodes.cfg` file (SQLSTATE 42729).
- Each *db-partition-number* listed in the ON DBPARTITIONNUMS clause must be for a unique partition (SQLSTATE 42728).
- A valid partition number is between 0 and 999 inclusive (SQLSTATE 42729).
- A partition cannot appear in both the ADD and DROP clauses (SQLSTATE 42728).
- There must be at least one partition remaining in the database partition group. The last partition cannot be dropped from a database partition group (SQLSTATE 428C0).
- If neither the LIKE DBPARTITIONNUM clause nor the WITHOUT TABLESPACES clause is specified when adding a partition, the default is to use the lowest partition number of the existing partitions in the database

partition group (say it is *2*) and proceed as if LIKE DBPARTITIONNUM *2* had been specified. For an existing partition to be used as the default it must have containers defined for all the table spaces in the database partition group (column IN_USE of SYSCAT.DBPARTITIONGROUPDEF is not 'T').

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NODE can be specified in place of DBPARTITIONNUM
    - NODES can be specified in place of DBPARTITIONNUMS
    - NODEGROUP can be specified in place of DATABASE PARTITION GROUP
- When a partition is added to a database partition group, a catalog entry is made for the partition (see SYSCAT.DBPARTITIONGROUPDEF). The partitioning map is changed immediately to include the new partition along with an indicator (IN_USE) that the partition is in the partitioning map if either:
  - no table spaces are defined in the database partition group or
  - no tables are defined in the table spaces defined in the database partition group and the WITHOUT TABLESPACES clause was not specified.

  The partitioning map is not changed and the indicator (IN_USE) is set to indicate that the partition is not included in the partitioning map if either:
  - tables exist in table spaces in the database partition group or
  - table spaces exist in the database partition group and the WITHOUT TABLESPACES clause was specified.

  To change the partitioning map, the REDISTRIBUTE DATABASE PARTITION GROUP command must be used. This redistributes any data, changes the partitioning map, and changes the indicator. Table space containers need to be added before attempting to redistribute data if the WITHOUT TABLESPACES clause was specified.
- When a partition is dropped from a database partition group, the catalog entry for the partition (see SYSCAT.DBPARTITIONGROUPDEF) is updated. If there are no tables defined in the table spaces defined in the database partition group, the partitioning map is changed immediately to exclude the dropped partition and the entry for the partition in the database partition group is dropped. If tables exist, the partitioning map is not changed and the indicator (IN_USE) is set to indicate that the partition is waiting to be dropped. The REDISTRIBUTE DATABASE PARTITION GROUP command must be used to redistribute the data and drop the entry for the partition from the database partition group.

Example:

Assume that you have a six-partition database that has the following partitions: 0, 1, 2, 5, 7, and 8. Two partitions are added to the system with partition numbers 3 and 6.

- Assume that you want to add partitions 3 and 6 to a database partition group called MAXGROUP and have table space containers like those on partition 2. The statement is as follows:

```
ALTER DATABASE PARTITION GROUP MAXGROUP
  ADD DBPARTITIONNUMS (3,6)LIKE DBPARTITIONNUM 2
```

- Assume that you want to drop partition 1 and add partition 6 to database partition group MEDGROUP. You will define the table space containers separately for partition 6 using ALTER TABLESPACE. The statement is as follows:

```
ALTER DATABASE PARTITION GROUP MEDGROUP
  ADD DBPARTITIONNUM(6)WITHOUT TABLESPACES
  DROP DBPARTITIONNUM(1)
```

Related concepts:

- "Data partitioning across multiple partitions" in the *SQL Reference, Volume 1*

ALTER FUNCTION

The ALTER FUNCTION statement modifies the properties of an existing function.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- ALTERIN privilege on the schema of the function
- Definer of the function, as recorded in the DEFINER column of SYSCAT.ROUTINES

To alter the EXTERNAL NAME of a function, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SYSADM or DBADM authority
- CREATE_EXTERNAL_ROUTINE authority on the database

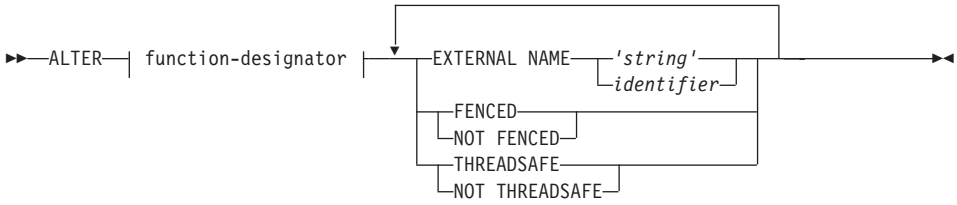
To alter a function to be not fenced, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SYSADM or DBADM authority
- CREATE_NOT_FENCED_ROUTINE authority on the database

To alter a function to be fenced, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:



Description:

function-designator
Uniquely identifies the function to be altered.

EXTERNAL NAME *'string'* **or** *identifier*
Identifies the name of the user-written code that implements the function. This option can only be specified when altering external functions (SQLSTATE 42849).

FENCED or **NOT FENCED**
Specifies whether the function is considered safe to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED). Most functions have the option of running as FENCED or NOT FENCED.

If a function is altered to be FENCED, the database manager insulates its internal resources (for example, data buffers) from access by the function. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION:
Use of NOT FENCED for functions that were not adequately coded, reviewed, and tested can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user-defined functions are used.

A function declared as NOT THREADSAFE cannot be altered to be NOT FENCED (SQLSTATE 42613).

If a function has any parameters defined AS LOCATOR, and was defined with the NO SQL option, the function cannot be altered to be FENCED (SQLSTATE 42613).

This option cannot be altered for LANGUAGE OLE or OLEDB functions (SQLSTATE 42849).

THREADSAFE or **NOT THREADSAFE**
Specifies whether the function is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the function is defined with LANGUAGE other than OLE and OLEDB:

- If the function is defined as THREADSAFE, the database manager can invoke the function in the same process as other routines. In general, to be threadsafe, a function should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED functions can be THREADSAFE.
- If the function is defined as NOT THREADSAFE, the database manager will never invoke the function in the same process as another routine. Only a fenced function can be NOT THREADSAFE (SQLSTATE 42613).

This option may not be altered for LANGUAGE OLE or OLEDB functions (SQLSTATE 42849).

Notes:

- It is not possible to alter a function that is in the SYSIBM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).
- Functions declared as LANGUAGE SQL, sourced functions, or template functions cannot be altered (SQLSTATE 42917).

Example:

The function MAIL() has been thoroughly tested. To improve its performance, alter the function to be not fenced.

```
ALTER FUNCTION MAIL() NOT FENCED
```

Related reference:

- "CREATE FUNCTION (OLE DB External Table)" on page 235
- "CREATE FUNCTION (External Scalar)" on page 190
- "CREATE FUNCTION (External Table)" on page 217
- "Common syntax elements" on page xi

ALTER METHOD

The ALTER METHOD statement modifies an existing method by changing the method body associated with the method.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

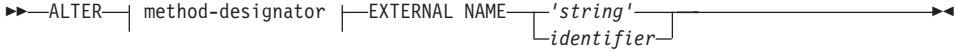
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATE_EXTERNAL_ROUTINE authority on the database, and at least one of:
  - ALTERIN privilege on the schema of the type
  - Definer of the type, as recorded in the DEFINER column of SYSCAT.DATATYPES

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:



Description:

method-designator
Uniquely identifies the method to be altered.

EXTERNAL NAME *'string'* **or** *identifier*
Identifies the name of the user-written code that implements the method. This option can only be specified when altering external methods (SQLSTATE 42849).

Notes:

- It is not possible to alter a method that is in the SYSIBM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).
- Methods declared as LANGUAGE SQL cannot be altered (SQLSTATE 42917).

- The specified method must have a body before it can be altered (SQLSTATE 42704).

Example:

Alter the method DISTANCE() in the structured type ADDRESS_T to use the library newaddresslib.

```
ALTER METHOD DISTANCE()
  FOR TYPE ADDRESS_T
  EXTERNAL NAME 'newaddresslib!distance2'
```

Related reference:

- "CREATE METHOD" on page 285

ALTER NICKNAME

The ALTER NICKNAME statement modifies the federated database's representation of a data source table or view by:

- Changing the local names of the table's or view's columns
- Changing the local data types of these columns
- Adding, changing, or deleting options for these columns

Invocation:

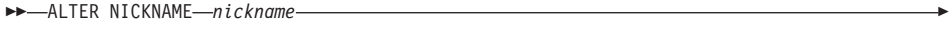
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

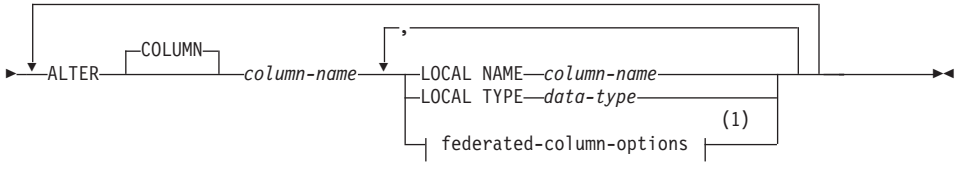
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

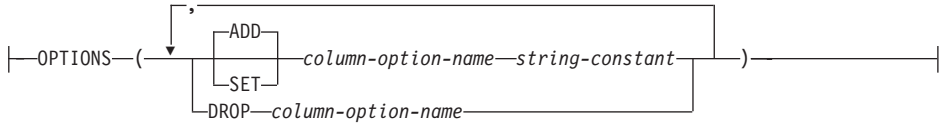
- SYSADM or DBADM authority
- ALTER privilege on the nickname specified in the statement
- CONTROL privilege on the nickname specified in the statement
- ALTERIN privilege on the schema, if the schema name of the nickname exists
- Definer of the nickname as recorded in the DEFINER column of the catalog view for the nickname

Syntax:





federated-column-options:



Notes:

1 If the user needs to specify the federated-column-options clause in addition to the LOCAL NAME parameter, the LOCAL TYPE parameter, or both, the user must specify the federated-column-options clause last.

Description:

nickname
Identifies the nickname for the data source table or view that contains the column specified after the COLUMN keyword. It must be a nickname described in the catalog.

ALTER COLUMN *column-name*
Names the column to be altered. The *column-name* is the federated server's current name for the column of the table or view at the data source. The *column-name* must identify an existing column of the data source table or view referenced by *nickname*. You cannot reference the same column name multiple times in the same ALTER NICKNAME statement. Enter all column options that you want to change under one ALTER COLUMN parameter, or use separate ALTER NICKNAME statements.

LOCAL NAME *column-name*
Is the new name by which the federated server is to reference the column identified by the ALTER COLUMN *column-name* parameter. This new name must be a valid DB2 identifier.

LOCAL TYPE *data-type*
Maps the specified column's data type to a local data type other than the one that it maps to now. The new type is denoted by *data-type*.

The data type cannot be LONG VARCHAR, LONG VARGRAPHIC, DATALINK, or a user-defined type.

OPTIONS
Indicates what column options are to be enabled, reset, or dropped for the column specified after the COLUMN keyword.

ADD
Enables a column option.

SET
Changes the setting of a column option.

column-option-name
Names a column option that is to be enabled or reset.

string-constant
Specifies the setting for *column-option-name* as a character string constant.

DROP *column-option-name*
Drops a column option.

Rules:

- If a view, SQL method, or SQL function has been created on a nickname, the ALTER NICKNAME statement cannot be used to change the local names or data types for the columns in the table or view that the nickname references (SQLSTATE 42601). The statement can be used, however, to enable, reset, or drop column options for these columns.

Notes:

- If ALTER NICKNAME is used to change the local name for a column in a table or view that a nickname references, queries of the column must reference it by its new name.
- A column option cannot be specified more than once in the same ALTER NICKNAME statement (SQLSTATE 42853). When a column option is enabled, reset, or dropped, any other column options that are in use are not affected.
- When the local specification of a column's data type is changed, the database manager invalidates any statistics (HIGH2KEY, LOW2KEY, and so on) gathered for that column.
- The ALTER NICKNAME statement cannot be processed within a unit of work (UOW) if the nickname referenced in this statement is already referenced by a SELECT statement in the same UOW (SQLSTATE 55007).

Examples:

Example 1: The nickname NICK1 references a DB2 Universal Database for AS/400 table called T1. Also, COL1 is the local name that references this table's first column, C1. Change the local name for C1 to NEWCOL.

```
ALTER NICKNAME NICK1
  ALTER COLUMN COL1
  LOCAL NAME NEWCOL
```

Example 2: The nickname EMPLOYEE references a DB2 Universal Database for OS/390 table called EMP. Also, SALARY is the local name that references

EMP_SAL, one of this table's columns. The column's data type, FLOAT, maps to the local data type, DOUBLE. Change the mapping so that FLOAT maps to DECIMAL (10, 5).

```
ALTER NICKNAME EMPLOYEE
  ALTER COLUMN SALARY
  LOCAL TYPE DECIMAL(10,5)
```

Example 3: Indicate that in an Oracle table, a column with the data type of VARCHAR doesn't have trailing blanks. The nickname for the table is NICK2, and the local name for the column is COL1.

```
ALTER NICKNAME NICK2
  ALTER COLUMN COL1
  OPTIONS ( ADD VARCHAR_NO_TRAILING_BLANKS 'Y' )
```

Related reference:

- "Column options for federated systems" in the *Federated Systems Guide*

ALTER PROCEDURE

The ALTER PROCEDURE statement modifies an existing procedure by changing the properties of the procedure.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- ALTERIN privilege on the schema of the procedure
- Definer of the procedure, as recorded in the DEFINER column of SYSCAT.ROUTINES

To alter the EXTERNAL NAME of a procedure, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SYSADM or DBADM authority
- CREATE_EXTERNAL_ROUTINE authority on the database

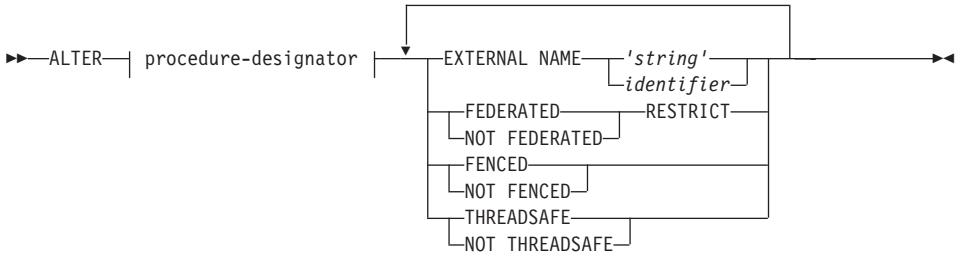
To alter a procedure to be not fenced, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SYSADM or DBADM authority
- CREATE_NOT_FENCED_ROUTINE authority on the database

To alter a procedure to be fenced, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:



Description:

procedure-designator
Uniquely identifies the procedure to be altered.

EXTERNAL NAME *'string'* **or** *identifier*
Identifies the name of the user-written code that implements the procedure. This option can only be specified when altering external procedures (SQLSTATE 42849).

FEDERATED or **NOT FEDERATED**
Specifies whether or not nicknames can be used.

If FEDERATED is specified, federated objects can be accessed from SQL statements in the procedure. A FEDERATED procedure that is also defined with the MODIFIES SQL DATA option cannot be used in a function, method, or trigger (SQLSTATE 42613). Statements within the procedure that access federated objects may be subject to special authorization rules.

If NOT FEDERATED is specified, federated objects cannot be used in any SQL statement in the procedure. Using a nickname will result in an error (SQLSTATE 55047).

The FEDERATED option cannot be altered for LANGUAGE OLE procedures (SQLSTATE 42849).

RESTRICT
The RESTRICT keyword enforces the rule that no object (other than a package) can be defined that depends on this procedure when altering the federated attribute (SQLSTATE 42893).

FENCED or **NOT FENCED**
Specifies whether the procedure is considered safe to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED). Most procedures have the option of running as FENCED or NOT FENCED.

If a procedure is altered to be FENCED, the database manager insulates its internal resources (for example, data buffers) from access by the

procedure. In general, a procedure running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION:
Use of NOT FENCED for procedures that were not adequately coded, reviewed, and tested can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED stored procedures are used.

A procedure declared as NOT THREADSAFE cannot be altered to be NOT FENCED (SQLSTATE 42613).

If a procedure has any parameters defined AS LOCATOR, and was defined with the NO SQL option, the procedure cannot be altered to be FENCED (SQLSTATE 42613).

This option cannot be altered for LANGUAGE OLE procedures (SQLSTATE 42849).

This option can only be specified when altering external procedures (SQLSTATE 42849).

THREADSAFE or **NOT THREADSAFE**
Specifies whether the procedure is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the procedure is defined with LANGUAGE other than OLE:

- If the procedure is defined as THREADSAFE, the database manager can invoke the procedure in the same process as other routines. In general, to be threadsafe, a procedure should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED procedures can be THREADSAFE.
- If the procedure is defined as NOT THREADSAFE, the database manager will never invoke the procedure in the same process as another routine. Only a fenced procedure can be NOT THREADSAFE (SQLSTATE 42613).

This option can only be specified when altering external procedures (SQLSTATE 42849).

This option may not be altered for LANGUAGE OLE procedures (SQLSTATE 42849).

Notes:

- It is not possible to alter a procedure that is in the SYSIBM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).
- Procedures declared as LANGUAGE SQL cannot be altered (SQLSTATE 42917).

Example:

Alter the procedure PARTS_ON_HAND() to be not fenced.

```
ALTER PROCEDURE PARTS_ON_HAND() NOT FENCED
```

Related reference:

- "CREATE PROCEDURE" on page 296

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:

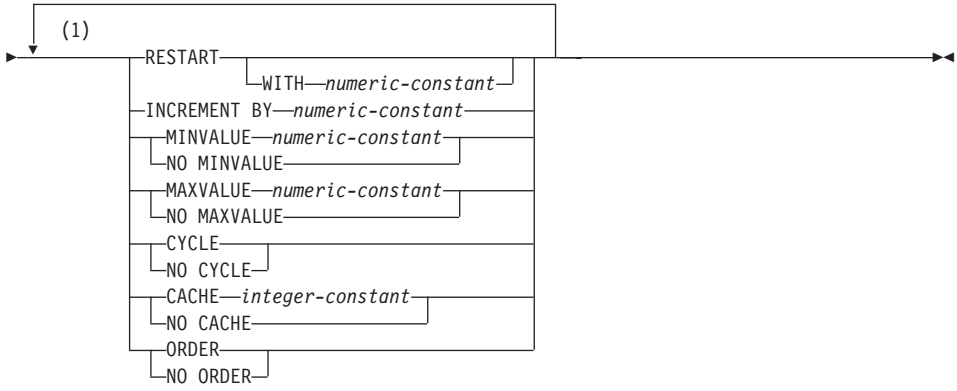
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- Definer of the sequence
- The ALTERIN privilege for the schema implicitly or explicitly specified
- SYSADM or DBADM authority

Syntax:



Notes:

1 The same clause must not be specified more than once.

Description:

sequence-name
Identifies the sequence that is to be changed. The name, including the implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists in the explicitly or implicitly specified schema, an error (SQLSTATE 42704) is returned. *sequence-name* must not be a sequence generated by the system for an identity column (SQLSTATE 428FB).

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 (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA).

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 (SQLSTATE 42815), and does not exceed the value of a large integer constant (SQLSTATE 42820), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA).

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.

MINVALUE or **NO 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.

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 (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be less than or equal to the maximum value (SQLSTATE 42815).

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.

MAXVALUE or **NO 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.

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 (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be greater than or equal to the minimum value (SQLSTATE 42815).

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.

CYCLE or **NO CYCLE**
Specifies whether the sequence should continue to generate values after reaching either its maximum or minimum value. The boundary of the sequence can be reached either with the next value landing exactly on the boundary condition, or by overshooting the value.

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 by DB2 for the sequence.

NO CYCLE
Specifies that values will not be generated for the sequence once the maximum or minimum value for the sequence has been reached.

CACHE or **NO CACHE**
Specifies whether to keep some preallocated values in memory for faster access. This is a performance and tuning option.

CACHE *integer-constant*
Specifies the maximum number of sequence values that are preallocated and kept in memory. Preallocating and storing values in the cache reduces synchronous I/O to the log when values are generated for the sequence.

In the event of a system failure, all cached sequence values that have not been used in committed statements are lost (that is, they will never be used). The value specified for the CACHE option is the maximum number of sequence values that could be lost in case of system failure.

The minimum value is 2 (SQLSTATE 42815).

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 synchronous I/O to the log.

ORDER or **NO ORDER**
Specifies whether the sequence numbers must be generated in order of request.

ORDER
Specifies that the sequence numbers are generated in order of request.

NO ORDER
Specifies that the sequence numbers do not need to be generated in order of request.

Notes:

- ***Compatibilities***

  - For compatibility with previous versions of DB2 and for consistency:
    - A comma can be used to separate multiple sequence options.
  - The following syntax is also supported:
    - NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER
- 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 to CYCLE, it is possible for sequence numbers to be duplicate values of ones generated by the sequence previously.

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. Another use would be to get results back where all the resulting rows are numbered:

```
ALTER SEQUENCE ORG_SEQ RESTART
  SELECT NEXTVAL FOR ORG_SEQ, ORG.* FROM ORG
```

Related samples:

- "DbSeq.java -- How to create, alter and drop a sequence in a database (JDBC)"
- "DbSeq.out -- HOW TO USE A SEQUENCE IN A DATABASE (JDBC)"

ALTER SERVER

The ALTER SERVER statement is used to:

- Modify the definition of a specific data source, or the definition of a category of data sources.
- Make changes in the configuration of a specific data source, or the configuration of a category of data sources—changes that will persist over multiple connections to the federated database.

(In this statement, the word SERVER and the parameter names that start with *server-* refer only to data sources in a federated system. They do not refer to the federated server in such a system, or to DRDA application servers.)

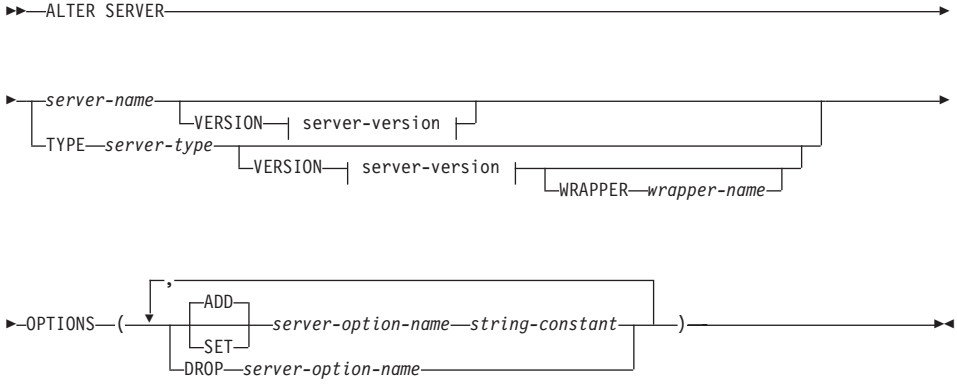
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

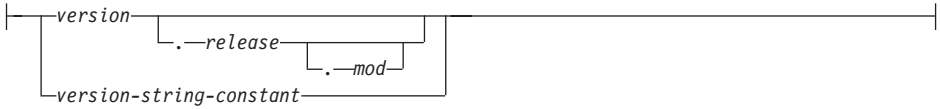
Authorization:

The authorization ID of the statement must include either SYSADM or DBADM authority on the federated database.

Syntax:



server-version:



Description:

server-name
Identifies the federated server's name for the data source to which the changes being requested are to apply. The data source must be one that is described in the catalog.

VERSION
After *server-name*, VERSION and its parameter specify a new version of the data source that *server-name* denotes.

version
Specifies the version number. *version* must be an integer.

release
Specifies the number of the release of the version denoted by *version*. *release* must be an integer.

mod
Specifies the number of the modification of the release denoted by *release*. *mod* must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release*, and, if applicable, *mod* (for example, '8.0.3').

TYPE *server-type*
Specifies the type of data source to which the changes being requested are to apply. The server type must be one that is listed in the catalog.

VERSION
After *server-type*, VERSION and its parameter specify the version of the data sources for which server options are to be enabled, reset, or dropped.

WRAPPER *wrapper-name*
Specifies the name of the wrapper that the federated server uses to interact with data sources of the type and version denoted by *server-type* and *server-version*. The wrapper must be listed in the catalog.

OPTIONS
Indicates what server options are to be enabled, reset, or dropped for the data source denoted by *server-name*, or for the category of data sources denoted by *server-type* and its associated parameters.

ADD
Enables a server option.

SET
Changes the setting of a server option.

server-option-name
Names a server option that is to be enabled or reset.

string-constant
Specifies the setting for *server-option-name* as a character string constant.

DROP *server-option-name*
Drops a server option.

Notes:

- This statement does not support the DBNAME and NODE server options (SQLSTATE 428EE).
- A server option cannot be specified more than once in the same ALTER SERVER statement (SQLSTATE 42853). When a server option is enabled, reset, or dropped, any other server options that are in use are not affected.
- An ALTER SERVER statement within a given unit of work (UOW) cannot be processed under either of the following conditions:
  - The statement references a single data source, and the UOW already includes a SELECT statement that references a nickname for a table or view within this data source (SQLSTATE 55007).
  - The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes a SELECT statement that references a nickname for a table or view within one of these data sources (SQLSTATE 55007).
- If the server option is set to one value for a type of data source, and set to another value for an instance of the type, the second value overrides the first one for the instance. For example, assume that PLAN_HINTS is set to 'Y' for server type ORACLE, and to 'N' for an Oracle data source named DELPHI. This configuration causes plan hints to be enabled at all Oracle data sources except DELPHI.

Examples:

Example 1: Ensure that when authorization IDs are sent to your Oracle 8.0.3 data sources, the case of the IDs will remain unchanged. Also, assume that these data sources have started to run on an upgraded CPU that's half as fast as your local CPU. Inform the optimizer of this statistic.

```
ALTER SERVER
  TYPE ORACLE
  VERSION 8.0.3
  OPTIONS
     ( ADD FOLD_ID 'N',
     SET CPU_RATIO '2.0' )
```

Example 2: Indicate that a DB2 Universal Database for AS/400 Version 3.0 data source called SUNDIAL has been upgraded to Version 3.1.

```
ALTER SERVER SUNDIAL
  VERSION 3.1
```

Related concepts:

- "Distributed relational databases" in the *SQL Reference, Volume 1*

Related reference:

- "Server options for federated systems" in the *Federated Systems Guide*

ALTER TABLE

The ALTER TABLE statement modifies existing tables by:

- Adding one or more columns to a table
- Adding or dropping a primary key
- Adding or dropping one or more unique or referential constraints
- Adding or dropping one or more check constraint definitions
- Adding or dropping a drop table restriction
- Altering the length of a VARCHAR column
- Altering a reference type column to add a scope
- Altering the generation expression of a generated column
- Altering one or more check or referential constraints attributes
- Adding or dropping a partitioning key
- Changing table attributes such as the data capture option, pctfree, lock size, or append mode.
- Setting the table to not logged initially state.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- ALTER privilege on the table to be altered
- CONTROL privilege on the table to be altered
- ALTERIN privilege on the schema of the table
- SYSADM or DBADM authority.

To create or drop a foreign key, the privileges held by the authorization ID of the statement must include one of the following on the parent table:

- REFERENCES privilege on the table
- REFERENCES privilege on each column of the specified parent key
- CONTROL privilege on the table
- SYSADM or DBADM authority.

To drop a primary key or unique constraint 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 this parent key of T:

- ALTER privilege on the table
- CONTROL privilege on the table
- ALTERIN privilege on the schema of the table
- SYSADM or DBADM authority.

To alter a table to become a materialized query table (using a fullselect), the privileges held by the authorization ID of the statement must include at least one of the following:

- CONTROL on the table
- SYSADM or DBADM authority;

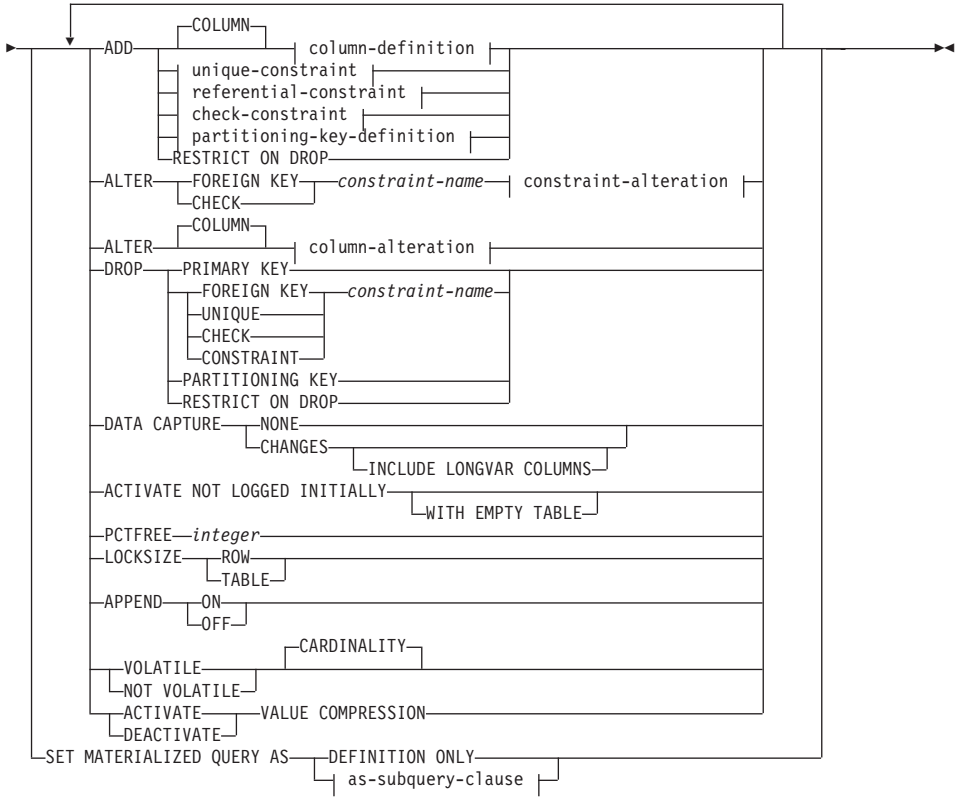
and at least one of the following, on each table or view identified in the fullselect:

- SELECT and ALTER privilege on the table or view
- CONTROL privilege on the table or view
- SELECT privilege on the table or view and ALTERIN privilege on the schema of the table or view
- SYSADM or DBADM authority.

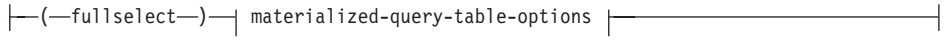
To alter a table so that it is no longer a materialized query table, the privileges held by the authorization ID of the statement must include at least one of the following, on each table or view identified in the fullselect used to define the materialized query table:

- ALTER privilege on the table or view
- CONTROL privilege on the table or view
- ALTERIN privilege on the schema of the table or view
- SYSADM or DBADM authority

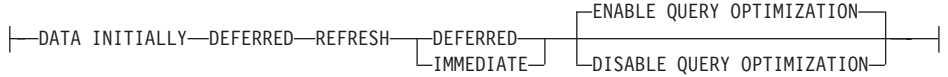
Syntax:



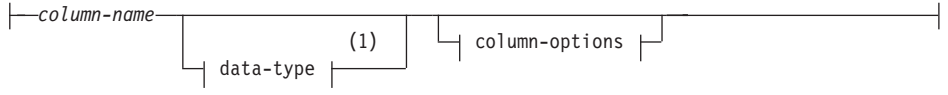
as-subquery-clause:



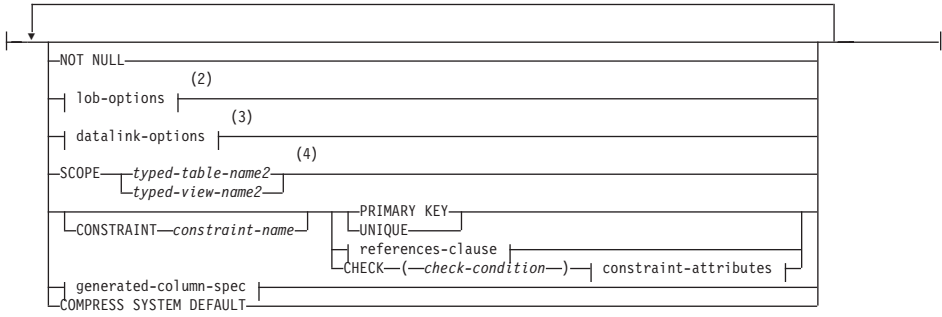
materialized-query-table-options:



column-definition:



column-options:



Notes:

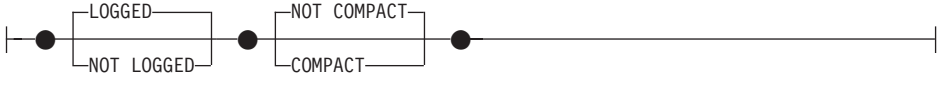
1 If the first column-option chosen is the generated-column-spec, then the data-type can be omitted and computed by the generation-expression.

2 The lob-options clause only applies to large object types (BLOB, CLOB and DBCLOB) and distinct types based on large object types.

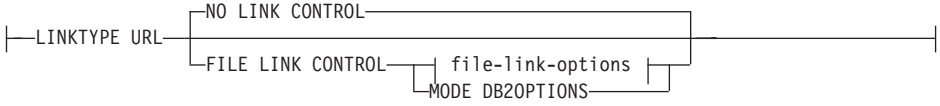
3 The datalink-options clause only applies to the DATALINK type and distinct types based on the DATALINK type.

4 The SCOPE clause only applies to the REF type.

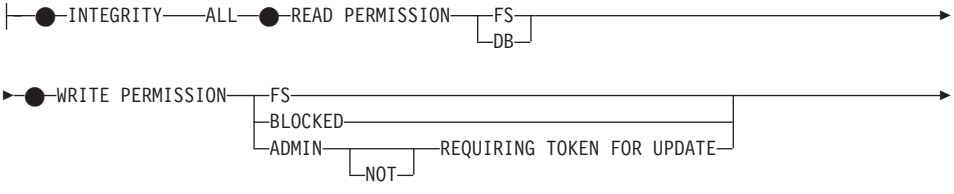
lob-options:



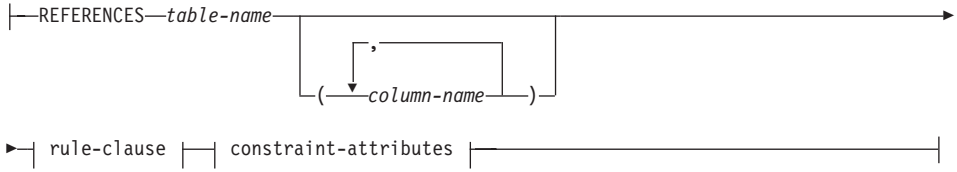
datalink-options:



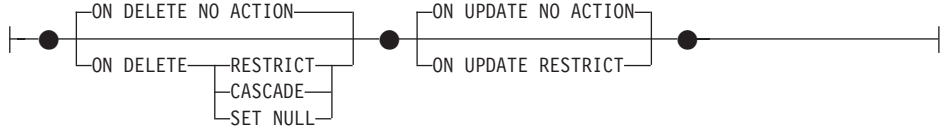
file-link-options:



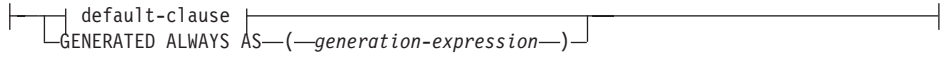
references-clause:



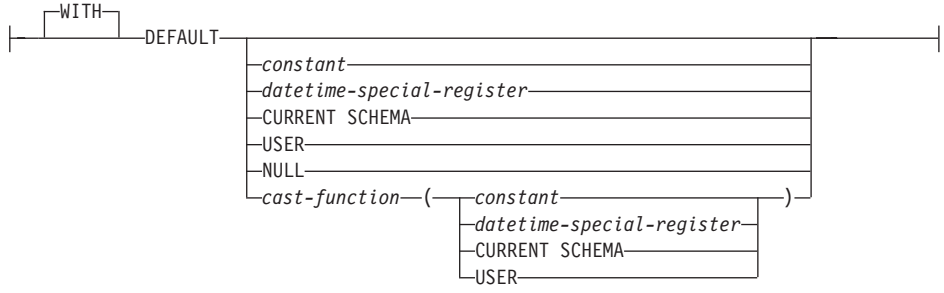
rule-clause:



generated-column-spec:

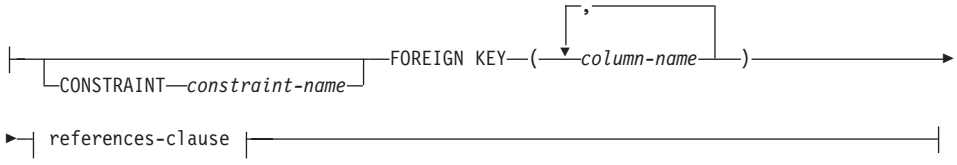


default-clause:

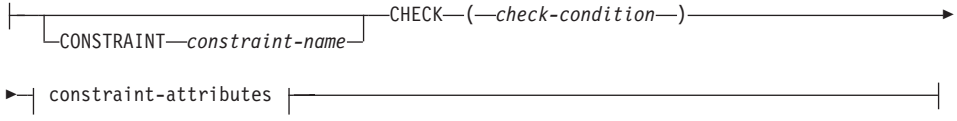


unique-constraint:

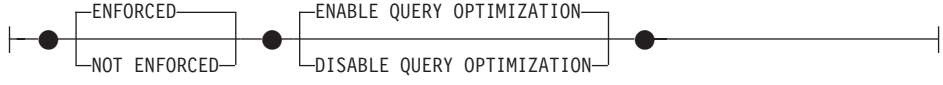
referential-constraint:



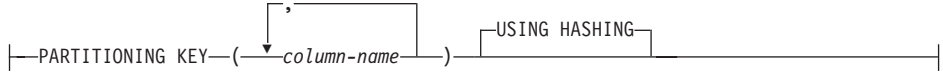
check-constraint:



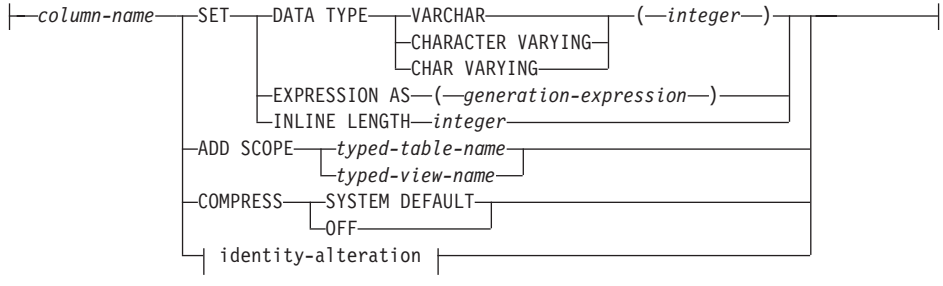
constraint-attributes:



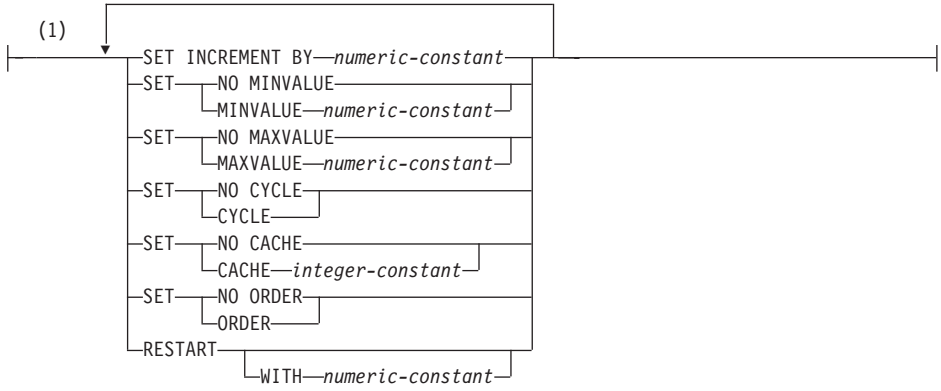
partitioning-key-definition:



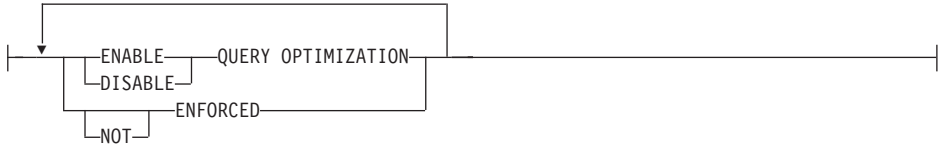
column-alteration:



identity-alteration:



constraint-alteration:



Notes:

1 The same clause must not be specified more than once.

Description:

table-name
Identifies the table to be changed. It must be a table described in the catalog and must not be a view or a catalog table. If *table-name* identifies a materialized query table, alterations are limited to setting the materialized query table to definition only, activating not logged initially, changing pctfree, locksize, append, or volatile. The *table-name* cannot be a nickname (SQLSTATE 42809) or a declared temporary table (SQLSTATE 42995).

SET MATERIALIZED QUERY AS
Allows alteration of the properties of a materialized query table.

DEFINITION ONLY
Change a materialized query table so that it is no longer considered a materialized query table. The table specified by *table-name* must be defined as a materialized query table that is not replicated (SQLSTATE 428EW). The definition of the columns of *table-name* is not changed but the table can no longer be used for query optimization and the REFRESH TABLE statement can no longer be used.

as-subquery-clause

Changes a regular table to a materialized query table for use during query optimization. The table specified by *table-name* must not:

- be previously defined as a materialized query table
- be a typed table
- have any constraints, unique indexes, or triggers defined
- be referenced in the definition of another materialized query table.

If table-name does not meet these criteria, an error is returned (SQLSTATE 428EW).

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 data types
- have the same column names in the same ordinal positions

as the result columns of *fullselect* (SQLSTATE 428EW). For details about specifying the *fullselect* for a materialized query table, see "CREATE TABLE". One additional restriction is that *table-name* cannot be directly or indirectly referenced in the fullselect.

materialized-query-table-options

Specifies the refreshable options for altering a materialized query table.

DATA INITIALLY DEFERRED

The data in the table must be validated using the REFRESH TABLE or SET INTEGRITY statement.

REFRESH

Indicates how the data in the table is maintained.

DEFERRED

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 the REFRESH TABLE statement is processed. Materialized query tables defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807).

IMMEDIATE

The changes made to the underlying tables as part of a DELETE, INSERT, or UPDATE are cascaded to the materialized query table. In this case, the content of the table, at any point-in-time, is the same as if the specified

subselect is processed. Materialized query tables defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807).

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.

ADD *column-definition*
Adds a column to the table. The table must not be a typed table (SQLSTATE 428DH). For all existing rows in the table, the value of the new column is set to its default value. The new column is the last column of the table; that is, if initially there are *n* columns, the added column is column *n*+1.

Adding the new column must not make the total byte count of all columns exceed the maximum record size.

column-name
Is the name of the column to be added to the table. The name cannot be qualified. Existing column names in the table cannot be used (SQLSTATE 42711).

data-type
Is one of the data types listed under "CREATE TABLE".

NOT NULL
Prevents the column from containing null values. The *default-clause* must also be specified (SQLSTATE 42601).

lob-options
Specifies options for LOB data types. See *lob-options* in "CREATE TABLE".

datalink-options
Specifies options for DATALINK data types. See *datalink-options* in "CREATE TABLE".

SCOPE
Specify a scope for a reference type column.

typed-table-name2
The name of a typed table. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name2* (SQLSTATE 428DM). No checking is done of the default value for *column-name* to ensure that the value actually references an existing row in *typed-table-name2*.

typed-view-name2
The name of a typed view. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name2* (SQLSTATE 428DM). No checking is done of the default value for *column-name* to ensure that the values actually references an existing row in *typed-view-name2*.

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that was already specified within the same ALTER TABLE statement, or as the name of any other existing constraint on the table (SQLSTATE 42710).

If the constraint name is not specified by the user, an 18-character identifier unique within the identifiers of the existing constraints defined on the table is generated by the system. (The identifier consists of ″SQL″ followed by a sequence of 15 numeric characters that are generated by a timestamp-based function.)

When used with a PRIMARY KEY or UNIQUE constraint, the *constraint-name* may be used as the name of an index that is created to support the constraint. See “Notes” on page 67 for details on index names associated with unique constraints.

PRIMARY KEY
This 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 were specified as a separate clause. The column cannot contain null values, so the NOT NULL attribute must also be specified (SQLSTATE 42831).

See PRIMARY KEY within the description of the *unique-constraint* below.

UNIQUE
This provides a shorthand method of defining a unique key 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 were specified as a separate clause.

See UNIQUE within the description of the *unique-constraint* below.

references-clause
This 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.

See *references-clause* in "CREATE TABLE".

CHECK (*check-condition***)**
This provides a shorthand method of defining a check constraint that applies to a single column. See *check-condition* in "CREATE TABLE".

generated-column-spec
For details on column-generation, see "CREATE TABLE".

default-clause
Specifies a default value for the column.

WITH
An optional keyword.

DEFAULT
Provides a default value in the event a value is not supplied on INSERT or is specified as DEFAULT on INSERT or UPDATE. If a specific default value is not specified following the DEFAULT keyword, the default value depends on the data type of the column as shown in Table 2. If a column is defined as a DATALINK or structured type, then a DEFAULT clause cannot be specified.

If a column is defined using a distinct type, then the default value of the column is the default value of the source data type cast to the distinct type.

Table 2. Default Values (when no value specified)

Data Type	Default Value
Numeric	0
Fixed-length character string	Blanks
Varying-length character string	A string of length 0
Fixed-length graphic string	Double-byte blanks
Varying-length graphic string	A string of length 0
Date	For existing rows, a date corresponding to January 1, 0001. For added rows, the current date.
Time	For existing rows, a time corresponding to 0 hours, 0 minutes, and 0 seconds. For added rows, the current time.
Timestamp	For existing rows, a date corresponding to January 1, 0001, and a time corresponding to 0 hours, 0 minutes, 0 seconds and 0 microseconds. For added rows, the current timestamp.

Table 2. Default Values (when no value specified) (continued)

Data Type	Default Value
Binary string (blob)	A string of length 0

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column.

Specific types of values that can be specified with the DEFAULT keyword are as follows.

constant
Specifies the constant as the default value 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 Chapter 3
- not be a floating-point constant unless the column is defined with a floating-point data type
- not have non-zero digits beyond the scale of the column data type if the constant is a decimal constant (for example, 1.234 cannot be the default for a DECIMAL(5,2) column)
- be expressed with no more than 254 characters including the quote characters, any introducer character such as the X for a hexadecimal constant, and characters from the fully qualified function name and parentheses when the constant is the argument of a *cast-function*.

datetime-special-register
Specifies the value of the datetime special register (CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be the data type that corresponds to the special register specified (for example, data type must be DATE when CURRENT DATE is specified). For existing rows, the value is the current date, current time or current timestamp when the ALTER TABLE statement is processed.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If CURRENT SCHEMA is specified, the data type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT

SCHEMA special register. For existing rows, the value of the CURRENT SCHEMA special register at the time the ALTER TABLE statement is processed.

USER
Specifies the value of the USER special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If USER is specified, the data type of the column must be a character string with a length not less than the length attribute of USER. For existing rows, the value is the authorization ID of the ALTER TABLE statement.

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
This form of a default value can only be used with columns defined as a distinct type, BLOB or datetime (DATE, TIME or TIMESTAMP) data type. For distinct type, with the exception of distinct types based on BLOB or datetime types, the name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type. For a distinct type based on a datetime type, where the default value is a constant, a function must be used and the name of the function must match the name of the source type of the distinct type with an implicit or explicit schema name of SYSIBM. For other datetime columns, the corresponding datetime function may also be used. For a BLOB or a distinct type based on BLOB, a function must be used and the name of the function must be BLOB with an implicit or explicit schema name of SYSIBM.

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. If the cast-function is BLOB, the constant must be a string constant.

datetime-special-register
Specifies CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The source type of the distinct

type of the column must be the data type that corresponds to the specified special register.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register. The data type of the source type of the distinct type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register. If the cast-function is BLOB, the length attribute must be at least 8 bytes.

USER
Specifies the USER special register. The data type of the source type of the distinct type of the column must be a string data type with a length of at least 8 bytes. If the cast-function is BLOB, the length attribute must be at least 8 bytes.

If the value specified is not valid, an error (SQLSTATE 42894) is raised.

COMPRESS SYSTEM DEFAULT
Specifies that system default values (that is, the default values used for the data types when no specific values are specified) are to be stored using minimal space. If the VALUE COMPRESSION clause is not specified, a warning is returned (SQLSTATE 01648) and system default values are not stored using minimal space.

Allowing system default values to be stored in this manner causes a slight performance penalty during insert and update operations on the column because of extra checking that is done.

The base data type must not be DATE, TIME, or TIMESTAMP (SQLSTATE 42842). If the base data type is a varying-length string, this clause is ignored. String values of length 0 are automatically compressed if a table has been set with VALUE COMPRESSION.

ADD *unique-constraint*
Defines a unique or primary key constraint. A primary key or unique constraint cannot be added to a table that is a subtable (SQLSTATE 429B3). If the table is a supertable at the top of the hierarchy, the constraint applies to the table and all its subtables.

CONSTRAINT *constraint-name*
Names the primary key or unique constraint. For more information, see *constraint-name* in "CREATE TABLE".

UNIQUE (*column-name***...,)**
Defines a unique key composed of the identified columns. The identified columns must be defined as NOT NULL. Each *column-name*

must identify a column of the table and the same column must not be identified more than once. The name cannot be qualified. The number of identified columns must not exceed 16, and the sum of their stored lengths must not exceed 1024. No LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, distinct type based on any of these types, or structured type may be used as part of a unique key, even if the length attribute of the column is small enough to fit within the 1024-byte limit (SQLSTATE 54008). The set of columns in the unique key cannot be the same as the set of columns of the primary key or another unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891.) Any existing values in the set of identified columns must be unique (SQLSTATE 23515).

A check is performed to determine if an existing index matches the unique key definition (ignoring any INCLUDE columns in the index). An index definition matches if it identifies the same set of columns without regard to the order of the columns or the direction (ASC/DESC) specifications. If a matching index definition is found, the description of the index is changed to indicate that it is required by the system and it is changed to unique (after ensuring uniqueness) if it was a non-unique index. If the table has more than one matching index, an existing unique index is selected (the selection is arbitrary). If no matching index is found, a unique index will automatically be created for the columns, as described in CREATE TABLE. See "Notes" on page 67 for details on index names associated with unique constraints.

PRIMARY KEY *...(column-name,)*

Defines a primary key composed of the identified columns. Each *column-name* must identify a column of the table, and the same column must not be identified more than once. The name cannot be qualified. The number of identified columns must not exceed 16 and the sum of their stored lengths must not exceed 1024. The table must not have a primary key and the identified columns must be defined as NOT NULL. No LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, distinct type based on any of these types, or structured type may be used as part of a primary key, even if the length attribute of the column is small enough to fit within the 1024-byte limit (SQLSTATE 54008). The set of columns in the primary key cannot be the same as the set of columns in a unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891.) Any existing values in the set of identified columns must be unique (SQLSTATE 23515).

A check is performed to determine if an existing index matches the primary key definition (ignoring any INCLUDE columns in the index). An index definition matches if it identifies the same set of

columns without regard to the order of the columns or the direction (ASC/DESC) specifications. If a matching index definition is found, the description of the index is changed to indicate that it is the primary index, as required by the system, and it is changed to unique (after ensuring uniqueness) if it was a non-unique index. If the table has more than one matching index, an existing unique index is selected (the selection is arbitrary). If no matching index is found, a unique index will automatically be created for the columns, as described in CREATE TABLE. See "Notes" on page 67 for details on index names associated with unique constraints.

Only one primary key can be defined on a table.

ADD *referential-constraint*
Defines a referential constraint. See *referential-constraint* in "CREATE TABLE".

ADD *check-constraint*
Defines a check constraint. See *check-constraint* in "CREATE TABLE".

ADD *partitioning-key-definition*
Defines a partitioning key. The table must be defined in a table space on a single-partition database partition group and must not already have a partitioning key. If a partitioning key already exists for the table, the existing key must be dropped before adding the new partitioning key.

A partitioning key cannot be added to a table that is a subtable (SQLSTATE 428DH).

PARTITIONING KEY (*column-name***...)**
Defines a partitioning key using the specified columns. Each *column-name* must identify a column of the table, and the same column must not be identified more than once. The name cannot be qualified. A column cannot be used as part of a partitioning key if the data type of the column is a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, DATALINK, distinct type on any of these types, or structured type.

USING HASHING
Specifies the use of the hashing function as the partitioning method for data distribution. This is the only partitioning method supported.

ADD RESTRICT ON DROP
Specifies that the table cannot be dropped, and that the table space that contains the table cannot be dropped.

ALTER FOREIGN KEY *constraint-name*
Alters the constraint attributes of the referential constraint *constraint-name*. The *constraint-name* must identify an existing referential constraint (SQLSTATE 42704).

ALTER CHECK *constraint-name*
Alters the constraint attributes of the check constraint *constraint-name*. The *constraint-name* must identify an existing referential constraint (SQLSTATE 42704).

constraint-alteration
Options for changing attributes associated with referential or check constraints.

ENFORCED
Change the constraint to ENFORCED. The constraint is enforced by the database manager during normal operations, such as insert, update, or delete.

NOT ENFORCED
Change the constraint to NOT ENFORCED. The constraint is not enforced by the database manager during normal operations, such as insert, update, or delete. This should only be specified if the table data is independently known to conform to the constraint.

ENABLE QUERY OPTIMIZATION
The constraint can be used for query optimization under appropriate circumstances.

DISABLE QUERY OPTIMIZATION
The constraint cannot be used for query optimization.

ALTER *column-alteration*
Alters the characteristics of a column.

column-name
Is the name of the column to be altered in the table. The *column-name* must identify an existing column of the table (SQLSTATE 42703). The name cannot be qualified.

SET DATA TYPE VARCHAR (*integer***)**
Increases the length of an existing VARCHAR column. CHARACTER VARYING or CHAR VARYING can be used as synonyms for the VARCHAR keyword. The data type of *column-name* must be VARCHAR and the current maximum length defined for the column must not be greater than the value for *integer* (SQLSTATE 42837). The value for *integer* can range up to 32 672. The table must not be a typed table (SQLSTATE 428DH).

Altering the column must not make the total byte count of all columns exceed the maximum record size (SQLSTATE 54010). 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 1024 (SQLSTATE 54008).

The length of variable length columns that are part of any index, including primary and unique keys, defined when the registry variable DB2_INDEX_2BYTEVARLEN was set to ON, can be altered to a length greater than 255 bytes. The fact that a variable length column is involved in a foreign key does not prevent the length of that column from being altered to larger than 255 bytes, regardless of the registry variable setting. However, data with length greater than 255 cannot be inserted into the table unless the column in the corresponding primary key has length greater than 255 bytes, which is only possible if the primary key was created with the registry variable set to ON.

SET EXPRESSION AS (*generation-expression***)**
Changes the expression for the column to the specified *generation-expression*. SET EXPRESSION AS requires the table to be put in check pending state, using the SET INTEGRITY statement. After the ALTER TABLE statement, the SET INTEGRITY statement must be used to update and check all the values in that column against the new expression. The column must already be defined as a generated column based on an expression (SQLSTATE 42837), and must not have appeared in the DIMENSIONS clause of the table (SQLSTATE 42997). The generation-expression must conform to the same rules that apply when defining a generated column. The result data type of the generation-expression must be assignable to the data type of the column (SQLSTATE 42821).

SET INLINE LENGTH *integer*
Changes the inline length of an existing structured type column. The inline length indicates the maximum byte size of an instance of a structured type to store inline with the rest of the values in the row. Instances of structured types that cannot be stored inline are stored separately from the base table row, similar to the way that LOB values are handled.

The data type of *column-name* must be a structured type (SQLSTATE 42842).

The default inline length for a structured-type column is the inline length of its type (specified explicitly or by default in the CREATE TYPE statement). If the inline length of a structured type is less than 292, the value 292 is used for the inline length of the column.

The explicit inline length value can only be increased (SQLSTATE -1); must be at least 292; and cannot exceed 32672 (SQLSTATE 54010).

Altering the column must not make the total byte count of all columns exceed the maximum record size (SQLSTATE 54010).

Data that is already stored separately from the rest of the row will not be moved inline by this statement. To take advantage of the altered inline length of a structured type column, invoke the REORG command against the specified table after altering the inline length of its column.

ADD SCOPE
Add a scope to an existing reference type column that does not already have a scope defined (SQLSTATE 428DK). If the table being altered is a typed table, the column must not be inherited from a supertable (SQLSTATE 428DJ).

typed-table-name
The name of a typed table. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

typed-view-name
The name of a typed view. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

COMPRESS
Specifies whether or not default values for this column are to be stored more efficiently.

SYSTEM DEFAULT
Specifies that system default values (that is, the default values used for the data types when no specific values are specified) are to be stored using minimal space. If the table is not already set with the VALUE COMPRESSION attribute activated, a warning is returned (SQLSTATE 01648), and system default values are not stored using minimal space.

Allowing system default values to be stored in this manner causes a slight performance penalty during insert and update operations on the column because of the extra checking that is done.

Existing data in the column is not changed. Consider offline table reorganization to enable existing data to take advantage of storing system default values using minimal space.

OFF
Specifies that system default values are to be stored in the column as regular values. Existing data in the column is not changed. Offline reorganization is recommended to change existing data.

The base data type must not be DATE, TIME or TIMESTAMP (SQLSTATE 42842). If the base data type is a varying-length string, this clause is ignored. String values of length 0 are automatically compressed if a table has been set with VALUE COMPRESSION.

If the table being altered is a typed table, the column must not be inherited from a supertable (SQLSTATE 428DJ).

SET INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the identity column. The next value to be generated for the identity column will be determined from the last assigned value with the increment applied. The column must already be defined with the IDENTITY attribute (SQLSTATE 42837).

This value can be any positive or negative value that could be assigned to this column (SQLSTATE 42815), and does not exceed the value of a large integer constant (SQLSTATE 42820), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA).

If this value is negative, this is a descending sequence after the ALTER statement. If this value is 0 or positive, this is an ascending sequence after the ALTER statement.

SET NO MINVALUE or MINVALUE *numeric-constant*
Specifies the minimum value at which a descending identity column either cycles or stops generating values, or the value to which an ascending identity column cycles after reaching the maximum value. The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837).

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 of the column.

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 this column (SQLSTATE 42815, SQLCODE -846), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA, SQLCODE -336), but the value must be less than or equal to the maximum value (SQLSTATE 42815, SQLCODE -846).

SET NO MAXVALUE or MAXVALUE *numeric-constant*
Specifies the maximum value at which an ascending identity column either cycles or stops generating values, or the value to which a

descending identity column cycles after reaching the minimum value. The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837).

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 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 this column (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be greater than or equal to the minimum value (SQLSTATE 42815).

SET NO CYCLE or CYCLE
Specifies whether this identity column should continue to generate values after generating either its maximum or minimum value. The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837).

NO CYCLE
Specifies that values will not be generated for the identity column once the maximum or minimum value has been reached.

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, then after an ascending identity column reaches the maximum value, it generates its minimum value; or after a descending sequence 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 for an identity column. Although not required, if unique values are desired, a single-column unique index defined using the identity column will ensure uniqueness. If a unique index exists on such an identity column and a non-unique value is generated, an error occurs (SQLSTATE 23505).

SET NO CACHE or CACHE *integer-constant*
Specifies whether to keep some pre-allocated values in memory for faster access. This is a performance and tuning option. The column must already be defined with the IDENTITY attribute (SQLSTATE 42837).

NO CACHE
Specifies that values for the identity column are not to be pre-allocated. In a data sharing environment, if the identity values *must* be generated in order of request, the NO CACHE option must be used.

When this option is specified, the values of the identity column are not stored in the cache. In this case, every request for a new identity value results in synchronous I/O to the log.

CACHE *integer-constant*
Specifies how many values of the identity sequence are pre-allocated and kept in memory. When values are generated for the identity column, pre-allocating and storing values in the cache reduces synchronous I/O to the log.

If a new value is needed for the identity column and there are no unused values available in the cache, the allocation of the value requires waiting for I/O to the log. However, when a new value is needed for the identity column and there is an unused value in the cache, the allocation of that identity value can happen more quickly by avoiding the I/O to the log.

In the event of a database deactivation, either normally or due to a system failure, all cached sequence values that have not been used in committed statements are lost (that is, they will never be used). The value specified for the CACHE option is the maximum number of values for the identity column that could be lost in case of system failure.

The minimum value is 2 (SQLSTATE 42815).

SET NO ORDER or ORDER
Specifies whether the identity column values must be generated in order of request. The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837).

NO ORDER
Specifies that the identity column values do not need to be generated in order of request.

ORDER
Specifies that the identity column values must be generated in order of request.

RESTART or RESTART WITH *numeric-constant*
Resets the state of the sequence associated with the identity column. If WITH *numeric-constant* is not specified, the sequence for the identity

column is restarted at the value that was specified, either implicitly or explicitly, as the starting value when the identity column was originally created.

The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837). RESTART does *not* change the original START WITH value.

The *numeric-constant* is an exact numeric constant that can be any positive or negative value that could be assigned to this column (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA). The *numeric-constant* will be used as the next value for the column.

DROP PRIMARY KEY
Drops the definition of the primary key and all referential constraints dependent on this primary key. The table must have a primary key.

DROP FOREIGN KEY *constraint-name*
Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint. For information on implications of dropping a referential constraint see "Notes" on page 67.

DROP UNIQUE *constraint-name*
Drops the definition of the unique constraint *constraint-name* and all referential constraints dependent on this unique constraint. The *constraint-name* must identify an existing UNIQUE constraint. For information on implications of dropping a unique constraint, see "Notes" on page 67.

DROP CHECK *constraint-name*
Drops the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint defined on the table.

DROP CONSTRAINT *constraint-name*
Drops the constraint *constraint-name*. The *constraint-name* must identify an existing check constraint, referential constraint, primary key or unique constraint defined on the table. For information on implications of dropping a constraint, see "Notes" on page 67.

DROP PARTITIONING KEY
Drops the partitioning key. The table must have a partitioning key and must be in a table space defined on a single-partition database partition group.

DROP RESTRICT ON DROP
Removes the restriction on dropping the table, and the table space that contains the table.

DATA CAPTURE
Indicates whether extra information for data replication is to be written to the log.

If the table is a typed table, then this option is not supported (SQLSTATE 428DH for root tables or 428DR for other subtables).

NONE
Indicates that no extra information will be logged.

CHANGES
Indicates that extra information regarding SQL changes to this table will be written to the log. This option is required if this table will be replicated and the Capture program is used to capture changes for this table from the log.

If the table is defined to allow data on a partition other than the catalog partition (multiple partition database partition group or database partition group with partition other than the catalog partition), then this option is not supported (SQLSTATE 42997).

If the schema name (implicit or explicit) of the table is longer than 18 bytes, this option is not supported (SQLSTATE 42997).

INCLUDE LONGVAR COLUMNS
Allows data replication utilities to capture changes made to LONG VARCHAR or LONG VARGRAPHIC columns. The clause may be specified for tables that do not have any LONG VARCHAR or LONG VARGRAPHIC columns since it is possible to ALTER the table to include such columns.

ACTIVATE NOT LOGGED INITIALLY
Activates the NOT LOGGED INITIALLY attribute of the table for this current unit of work.

Any changes made to the table by an INSERT, DELETE, UPDATE, CREATE INDEX, DROP INDEX, or ALTER TABLE in the same unit of work after the table is altered by this statement are not logged. Any changes made to the system catalog by the ALTER statement in which the NOT LOGGED INITIALLY attribute is activated are logged. Any subsequent changes made in the same unit of work to the system catalog information are logged.

At the completion of the current unit of work, the NOT LOGGED INITIALLY attribute is deactivated and all operations that are done on the table in subsequent units of work are logged.

If using this feature to avoid locks on the catalog tables while inserting data, it is important that only this clause be specified on the ALTER TABLE statement. Use of any other clause in the ALTER TABLE statement will result in catalog locks. If no other clauses are specified for the ALTER

TABLE statement, then only a SHARE lock will be acquired on the system catalog tables. This can greatly reduce the possibility of concurrency conflicts for the duration of time between when this statement is executed and when the unit of work in which it was executed is ended.

If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

For more information about the NOT LOGGED INITIALLY attribute, see the description of this attribute in "CREATE TABLE".

Note: If non-logged activity occurs against a table that has the NOT LOGGED INITIALLY attribute activated, and if a statement fails (causing a rollback), or a ROLLBACK TO SAVEPOINT is executed, the entire unit of work is rolled back (SQL1476N). Furthermore, the table for which the NOT LOGGED INITIALLY attribute was activated is marked inaccessible after the rollback has occurred and can only be dropped. Therefore, the opportunity for errors within the unit of work in which the NOT LOGGED INITIALLY attribute is activated should be minimized.

WITH EMPTY TABLE
Causes all data currently in table to be removed. Once the data has been removed, it cannot be recovered except through use of the RESTORE facility. If the unit of work in which this Alter statement was issued is rolled back, the table data will NOT be returned to its original state.

When this action is requested, no DELETE triggers defined on the affected table are fired. Any indexes that exist on the table are also emptied.

PCTFREE *integer*
Indicates what percentage of each page to leave as free space during load or reorganization. The value of *integer* can range from 0 to 99. The first row on each page is added without restriction. When additional rows are added, at least *integer* percent of free space is left on each page. The PCTFREE value is considered only by the LOAD and REORGANIZE TABLE utilities. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

LOCKSIZE
Indicates the size (granularity) of locks used when the table is accessed. Use of this option in the table definition will not prevent normal lock escalation from occurring. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

ROW
Indicates the use of row locks. This is the default lock size when a table is created.

TABLE
Indicates the use of table locks. This means that the appropriate share or exclusive lock is acquired on the table and intent locks (except intent none) are not used. Use of this value may improve the performance of queries by limiting the number of locks that need to be acquired. However, concurrency is also reduced since all locks are held over the complete table.

APPEND
Indicates whether data is appended to the end of the table data or placed where free space is available in data pages. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

ON
Indicates that table data will be appended and information about free space on pages will not be kept. The table must not have a clustered index (SQLSTATE 428CA).

OFF
Indicates that table data will be placed where there is available space. This is the default when a table is created.

The table should be reorganized after setting APPEND OFF since the information about available free space is not accurate and may result in poor performance during insert.

VOLATILE
This indicates to the optimizer that the cardinality of table *table-name* can vary significantly at run time, from empty to quite large. To access *table-name* the optimizer will use an index scan rather than a table scan, regardless of the statistics, if that index is index-only (all columns referenced are in the index) or that index is able to apply a predicate in the index scan. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

NOT VOLATILE
This indicates to the optimizer that the cardinality of *table-name* is not volatile. Access Plans to this table will continue to be based on the existing statistics and on the optimization level in place.

CARDINALITY
An optional key word to indicate that it is the number of rows in the table that is volatile and not the table itself.

VALUE COMPRESSION
Specifies whether or not NULL and 0-length data values are to be stored more efficiently for most data types. This also determines the row format that is to be used. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

ACTIVATE
Specifies that 0-length data values for columns whose data type is BLOB, CLOB, DBCLOB, LONG VARCHAR, or LONG VARGRAPHIC are to be stored using minimal space. Each NULL value is stored without using an additional byte. The row format that is used to support this determines the byte counts for each data type, and tends to cause data fragmentation during updates. The new row format (specified for a column through the COMPRESS SYSTEM DEFAULT option) also allows system default values for the column to be stored more efficiently.

DEACTIVATE
Specifies that NULL values are to be stored with space set aside for possible future updates. This space is not set aside for varying-length columns. The row format used determines the byte counts for each data type. It also does not support efficient storage of system default values for a column. If columns already exist with the COMPRESS SYSTEM DEFAULT attribute, a warning is returned (SQLSTATE 01648).

An update operation will cause an existing row to be changed to the new row format. Offline table reorganization is recommended to improve the performance of update operations on existing rows.

Rules:

- Any unique or primary key constraint defined on the table must be a superset of the partitioning key, if there is one (SQLSTATE 42997).
- Primary or unique keys cannot be subsets of dimensions (SQLSTATE 429BE).
- A column can only be referenced in one ADD or ALTER COLUMN clause in a single ALTER TABLE statement (SQLSTATE 42711).
- A column length cannot be altered if the table has any materialized query tables that are dependent on the table (SQLSTATE 42997).
- Before adding a generated column, the table must be set into the check-pending state, using the SET INTEGRITY statement (SQLSTATE 55019).

Notes:

- Altering a table to a materialized query table will put the table in check-pending state. If the table is defined as REFRESH IMMEDIATE, the table must be taken out of check-pending state before INSERT, DELETE, or UPDATE commands can be invoked on the table referenced by the fullselect. The table can be taken out of check-pending state by using REFRESH TABLE or SET INTEGRITY, with the IMMEDIATE CHECKED option, to completely refresh the data in the table based on the fullselect. If the data in the table accurately reflects the result of the fullselect, the IMMEDIATE UNCHECKED option of SET INTEGRITY can be used to take the table out of check-pending state.
- Altering a table to change it to a REFRESH IMMEDIATE materialized query table will cause any packages with INSERT, DELETE, or UPDATE usage on the table referenced by the fullselect to be invalidated.
- Altering a table to change from a materialized query table to a regular table (DEFINITION ONLY) will cause any packages dependent on the table to be invalidated.
- If a deferred materialized query table is associated with a staging table, the staging table will be dropped if the materialized query table is altered to a regular table.
- ADD column clauses are processed prior to all other clauses. Other clauses are processed in the order that they are specified.
- Any columns added via ALTER TABLE will not automatically be added to any existing view of the table.
- When an index is automatically created for a unique or primary key constraint, the database manager will try to use the specified constraint name as the index name with a schema name that matches the schema name of the table. If this matches an existing index name or no name for the constraint was specified, the index is created in the SYSIBM schema with a system-generated name formed of ″SQL″ followed by a sequence of 15 numeric characters generated by a timestamp based function.
- Any table that may be involved in a DELETE operation on table T is said to be *delete-connected* to T. Thus, a table is delete-connected to T if it is a dependent of T or it is a dependent of a table in which deletes from T cascade.
- A package has an insert (update/delete) usage on table T if records are inserted into (updated in/deleted from) T either directly by a statement in the package, or indirectly through constraints or triggers executed by the package on behalf of one of its statements. Similarly, a package has an update usage on a column if the column is modified directly by a statement in the package, or indirectly through constraints or triggers executed by the package on behalf of one of its statements.
- Any changes to primary key, unique keys, or foreign keys may have the following effect on packages, indexes, and other foreign keys.

- If a primary key or unique key is added:
  - There is no effect on packages, foreign keys, or existing unique keys. (If the primary or unique key uses an existing unique index that was created in a previous version and has not been converted to support deferred uniqueness, the index is converted, and packages with update usage on the associated table are invalidated.)
- If a primary key or unique key is dropped:
  - The index is dropped if it was automatically created for the constraint. Any packages dependent on the index are invalidated.
  - The index is set back to non-unique if it was converted to unique for the constraint and it is no longer system-required. Any packages dependent on the index are invalidated.
  - The index is set to no longer system required if it was an existing unique index used for the constraint. There is no effect on packages.
  - All dependent foreign keys are dropped. Further action is taken for each dependent foreign key, as specified in the next item.
- If a foreign key is added, dropped, or altered from NOT ENFORCED to ENFORCED (or ENFORCED to NOT ENFORCED):
  - All packages with an insert usage on the object table are invalidated.
  - All packages with an update usage on at least one column in the foreign key are invalidated.
  - All packages with a delete usage on the parent table are invalidated.
  - All packages with an update usage on at least one column in the parent key are invalidated.
- If a foreign key is altered from ENABLE QUERY OPTIMIZATION to DISABLE QUERY OPTIMIZATION:
  - All packages with dependencies on the constraint for optimization purposes are invalidated.
- Adding a column to a table will result in invalidation of all packages with insert usage on the altered table. If the added column is the first user-defined structured type column in the table, packages with DELETE usage on the altered table will also be invalidated.
- Adding a check or referential constraint to a table that already exists and that is not in check pending state, or altering the existing check or referential constraint from NOT ENFORCED to ENFORCED on an existing table that is not in check pending state will cause the existing rows in the table to be immediately evaluated against the constraint. If the verification fails, an error (SQLSTATE 23512) is raised. If a table is in check pending state, adding a check or referential constraint, or altering a constraint from NOT ENFORCED to ENFORCED will not immediately lead to the enforcement of the constraint. Instead, the corresponding

constraint type flags used in the check pending operation will be updated. Issue the SET INTEGRITY statement to begin enforcing the constraint.

- Adding, altering, or dropping a check constraint will result in invalidation of all packages with either an insert usage on the object table, an update usage on at least one of the columns involved in the constraint, or a select usage exploiting the constraint to improve performance.
- Adding a partitioning key will result in invalidation of all packages with an update usage on at least one of the columns of the partitioning key.
- A partitioning key that was defined by default as the first column of the primary key is not affected by dropping the primary key and adding a different primary key.
- Altering a column to increase the length will invalidate all packages that reference the table (directly or indirectly through a referential constraint or trigger) with the altered column.
- Altering a column to increase the length will regenerate views (except typed views) that are dependent on the table. If an error occurs while regenerating a view, an error is returned (SQLSTATE 56098). Any typed views that are dependent on the table are marked inoperative.
- Altering a column to increase the length may cause errors (SQLSTATE 54010) in processing triggers when a statement that would involve the trigger is prepared or bound. This may occur when row length based on the sum of the lengths of the transition variables and transition table columns is too long. If such a trigger were dropped a subsequent attempt to create it would result in an error (SQLSTATE 54040).
- VARCHAR and VARGRAPHIC columns that have been altered to be greater than 4000 and 2000 respectively should not be used as input parameters in functions in the SYSFUN schema (SQLSTATE 22001).
- Altering a structured type column to increase the inline length will invalidate all packages that reference the table, either directly or indirectly through a referential constraint or trigger.
- Altering a structured type column to increase the inline length will regenerate views that are dependent on the table.
- Changing the LOCKSIZE for a table will result in invalidation of all packages that have a dependency on the altered table.
- The ACTIVATE NOT LOGGED INITIALLY clause can not be used when DATALINK columns with the FILE LINK CONTROL attribute are being added to the table (SQLSTATE 42613).
- Changing VOLATILE or NOT VOLATILE CARDINALITY will result in invalidation of all packages that have a dependency on the altered table.
- Replication customers should take caution when increasing the length of VARCHAR columns. The change data table associated with an

application table might already be at or near the DB2 rowsize limit. The change data table should be altered before the application table, or the two should be altered within the same unit of work, to ensure that the alteration can be completed for both tables. Consideration should be given for copies, which may also be at or near the rowsize limit, or reside on platforms which lack the feature to increase the length of an existing column.

If the change data table is not altered before the Capture program processes log records with the increased VARCHAR column length, the Capture program will likely fail. If a copy containing the VARCHAR column is not altered before the subscription maintaining the copy runs, the subscription will likely fail.

– ***Compatibilities***
  - For compatibility with previous versions of DB2:
    • The ADD keyword is optional for:
      – Unnamed PRIMARY KEY constraints
      – Unnamed referential constraints
      – Referential constraints whose name follows the phrase FOREIGN KEY
    • The CONSTRAINT keyword can be omitted from a *column-definition* defining a references-clause
    • *constraint-name* can be specified following FOREIGN KEY (without the CONSTRAINT keyword)
    • SET SUMMARY AS can be specified in place of SET MATERIALIZED QUERY AS
  - For compatibility with previous versions of DB2 and for consistency:
    • A comma can be used to separate multiple options in the *identity-alteration* clause.
  - The following syntax is also supported:
    • NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER

Examples:

Example 1: Add a new column named RATING, which is one character long, to the DEPARTMENT table.

```
ALTER TABLE DEPARTMENT
   ADD RATING CHAR(1)
```

Example 2: Add a new column named SITE_NOTES to the PROJECT table. Create SITE_NOTES as a varying-length column with a maximum length of

1000 characters. The values of the column do not have an associated character set and therefore should not be translated.

```
ALTER TABLE PROJECT
   ADD SITE_NOTES  VARCHAR(1000) FOR BIT DATA
```

Example 3: Assume a table called EQUIPMENT exists defined with the following columns:

```
Column Name         Data Type
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. DEPTNO is the primary key of 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
```

Also, an additional column is needed to allow the recording of the quantity associated with this equipment record. Unless otherwise specified, the EQUIP_QTY column should have a value of 1 and must never be null.

```
ALTER TABLE EQUIPMENT
  ADD COLUMN EQUIP_QTY
  SMALLINT NOT NULL DEFAULT 1
```

Example 4: Alter table EMPLOYEE. Add the check constraint named REVENUE defined so that each employee must make a total of salary and commission greater than $30,000.

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT REVENUE
  CHECK (SALARY + COMM > 30000)
```

Example 5: Alter table EMPLOYEE. Drop the constraint REVENUE which was previously defined.

```
ALTER TABLE EMPLOYEE
  DROP CONSTRAINT REVENUE
```

Example 6: Alter a table to log SQL changes in the default format.

```
ALTER TABLE SALARY1
  DATA CAPTURE NONE
```

Example 7: Alter a table to log SQL changes in an expanded format.

```
ALTER TABLE SALARY2
  DATA CAPTURE CHANGES
```

Example 8: Alter the EMPLOYEE table to add 4 new columns with default values.

```
ALTER TABLE EMPLOYEE
   ADD COLUMN HEIGHT MEASURE   DEFAULT MEASURE(1)
   ADD COLUMN BIRTHDAY BIRTHDATE DEFAULT DATE('01-01-1850')
   ADD COLUMN FLAGS BLOB(1M)  DEFAULT BLOB(X'01')
   ADD COLUMN PHOTO PICTURE   DEFAULT BLOB(X'00')
```

The default values use various function names when specifying the default. Since MEASURE is a distinct type based on INTEGER, the MEASURE function is used. The HEIGHT column default could have been specified without the function since the source type of MEASURE is not BLOB or a datetime data type. Since BIRTHDATE is a distinct type based on DATE, the DATE function is used (BIRTHDATE cannot be used here). For the FLAGS and PHOTO columns the default is specified using the BLOB function even though PHOTO is a distinct type. To specify a default for BIRTHDAY, FLAGS and PHOTO columns, a function must be used because the type is a BLOB or a distinct type sourced on a BLOB or datetime data type.

Example 9: A table called CUSTOMERS is defined with the following columns:

```
Column Name         Data Type
BRANCH_NO           SMALLINT
CUSTOMER_NO         DECIMAL(7)
CUSTOMER_NAME       VARCHAR(50)
```

In this table, the primary key is made up of the BRANCH_NO and CUSTOMER_NO columns. To partition the table, you will need to create a partitioning key for the table. The table must be defined in a table space on a single-node database partition group. The primary key must be a superset of the partitioning columns: at least one of the columns of the primary key must be used as the partitioning key. Make BRANCH_NO the partitioning key as follows:

```
ALTER TABLE CUSTOMERS
  ADD PARTITIONING KEY (BRANCH_NO)
```

Related reference:

- "ALTER TYPE (Structured)" on page 83
- "CREATE TABLE" on page 332

Related samples:

- "dbrecov.sqc -- How to recover a database (C)"
- "tbconstr.sqc -- How to create, use, and drop constraints (C)"

- “dbrecov.sqC -- How to recover a database (C++)”
- “dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)”
- “tbconstr.sqC -- How to create, use, and drop constraints (C++)”
- “TbGenCol.java -- How to use generated columns (JDBC)”

ALTER TABLESPACE

The ALTER TABLESPACE statement is used to modify an existing tablespace in the following ways:

- Add a container to, or drop a container from a DMS tablespace; that is, a tablespace created with the MANAGED BY DATABASE option.
- Modify the size of a container in a DMS tablespace.
- Add a container to an SMS table space on a partition that currently has no containers.
- Modify the PREFETCHSIZE setting for a tablespace.
- Modify the BUFFERPOOL used for tables in the tablespace.
- Modify the OVERHEAD setting for a tablespace.
- Modify the TRANSFERRATE setting for a tablespace.

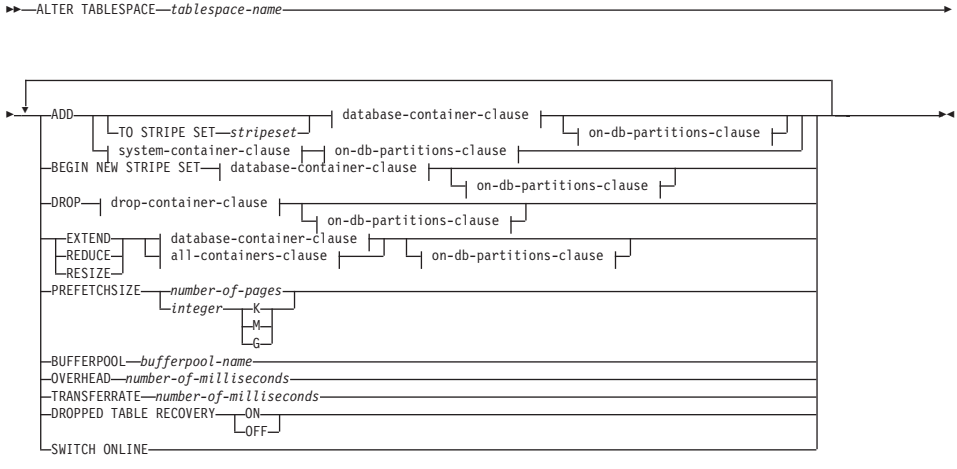
Invocation:

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

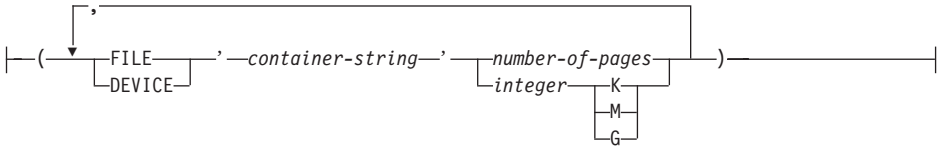
Authorization:

The authorization ID of the statement must have SYSCTRL or SYSADM authority.

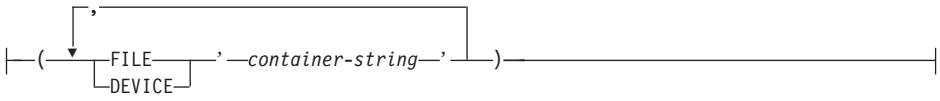
Syntax:



database-container-clause:

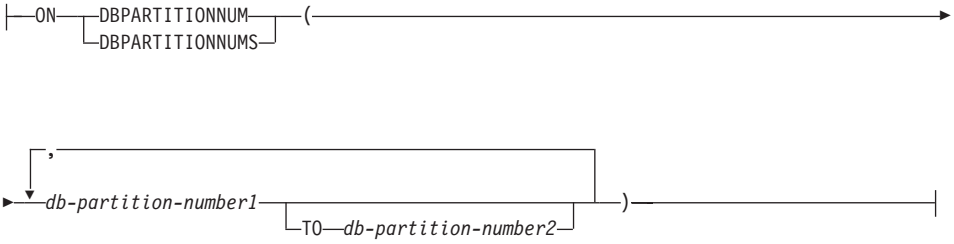


drop-container-clause:

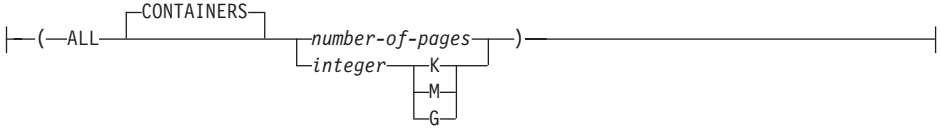


system-container-clause:

on-db-partitions-clause:



all-containers-clause:



Description:

tablespace-name
Names the tablespace. This is a one-part name. It is a long SQL identifier (either ordinary or delimited).

ADD
Specifies that one or more new containers are to be added to the tablespace.

TO STRIPE SET *stripeset*
Specifies that one or more new containers are to be added to the tablespace, and that they will be placed into the given stripe set.

BEGIN NEW STRIPE SET
Specifies that a new stripe set is to be created in the tablespace, and that one or more containers are to be added to this new stripe set. Containers that are subsequently added using the ADD option will be added to this new stripe set unless TO STRIPE SET is specified.

DROP
Specifies that one or more containers are to be dropped from the tablespace.

EXTEND
Specifies that existing containers are to be increased in size. The size specified is the size by which the existing container is increased. If the *all-containers-clause* is specified, all containers in the tablespace will increase by this size.

REDUCE
Specifies that existing containers are to be reduced in size. The size specified is the size by which the existing container is decreased. If the *all-containers-clause* is specified, all containers in the tablespace will decrease by this size.

RESIZE
Specifies that the size of existing containers is to be changed. The size specified is the new size for the container. If the *all-containers-clause* is specified, all containers in the tablespace will be changed to this size. If the operation affects more than one container, these containers must all either increase in size, or decrease in size. It is not possible to increase some while decreasing others (SQLSTATE 429BC).

database-container-clause
Adds one or more containers to a DMS tablespace. The tablespace must identify a DMS tablespace that already exists at the application server.

drop-container-clause
Drops one or more containers from a DMS tablespace. The tablespace must identify a DMS tablespace that already exists at the application server.

system-container-clause
Adds one or more containers to an SMS tablespace on the specified partitions. The tablespace must identify an SMS tablespace that already exists at the application server. There must not be any containers on the specified partitions for the tablespace. (SQLSTATE 42921).

on-db-partitions-clause
Specifies one or more partitions for the corresponding container operations.

all-containers-clause
Extends, reduces, or resizes all of the containers in a DMS tablespace. The tablespace must identify a DMS tablespace that already exists at the application server.

PREFETCHSIZE *number-of-pages*
Specifies the number of PAGESIZE pages that will be read from the tablespace when data prefetching is being performed. The prefetch size value can also be specified as an integer value followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages value for prefetch size. Prefetching reads in data needed by a query prior to it being referenced by the query, so that the query need not wait for I/O to be performed.

BUFFERPOOL *bufferpool-name*
The name of the buffer pool used for tables in this tablespace. The buffer pool must currently exist in the database (SQLSTATE 42704). The database partition group of the tablespace must be defined for the bufferpool (SQLSTATE 42735).

OVERHEAD *number-of-milliseconds*
Any numeric literal (integer, decimal, or floating point) that specifies the I/O controller overhead and disk seek and latency time, in milliseconds. The number should be an average for all containers that belong to the tablespace, if not the same for all containers. This value is used to determine the cost of I/O during query optimization.

TRANSFERRATE *number-of-milliseconds*
Any numeric literal (integer, decimal, or floating point) that specifies the time to read one page (4K or 8K) into memory, in milliseconds. The number should be an average for all containers that belong to the tablespace, if not the same for all containers. This value is used to determine the cost of I/O during query optimization.

DROPPED TABLE RECOVERY
Dropped tables in the specified tablespace may be recovered using the RECOVER DROPPED TABLE ON option of the ROLLFORWARD command.

SWITCH ONLINE
Tablespaces in OFFLINE state are brought online if the containers have become accessible. If the containers are not accessible an error is returned (SQLSTATE 57048).

Notes:

- *Compatibilities*

  For compatibility with versions earlier than Version 8, the keyword:
  - NODE can be substituted for DBPARTITIONNUM
  - NODES can be substituted for DBPARTITIONNUMS
- Default container operations are container operations that are specified in the ALTER TABLESPACE statement, but that are not explicitly directed to a specific database partition. These container operations are sent to any database partition that is not listed in the statement. If these default container operations are not sent to any database partition, because all database partitions are explicitly mentioned for a container operation, a warning is returned (SQLSTATE 1758W).
- Once space has been added or removed from a tablespace, and the transaction is committed, the contents of the tablespace may be rebalanced across the containers. Access to the tablespace is not restricted during rebalancing.
- If the tablespace is in OFFLINE state and the containers have become accessible, the user can disconnect all applications and connect to the database again to bring the tablespace out of OFFLINE state. Alternatively, SWITCH ONLINE option can bring the tablespace up (out of OFFLINE) while the rest of the database is still up and being used.
- If adding more than one container to a tablespace, it is recommended that they be added in the same statement so that the cost of rebalancing is incurred only once. An attempt to add containers to the same tablespace in separate ALTER TABLESPACE statements within a single transaction will result in an error (SQLSTATE 55041).
- Any attempts to extend, reduce, resize, or drop containers that do not exist will raise an error (SQLSTATE 428B2).
- When extending, reducing, or resizing a container, the container type must match the type that was used when the container was created (SQLSTATE 428B2).
- An attempt to change container sizes in the same tablespace, using separate ALTER TABLESPACE statements but within a single transaction, will raise an error (SQLSTATE 55041).
- In a partitioned database if more than one database partition resides on the same physical node, the same device or specific path cannot be specified for such database partitions (SQLSTATE 42730). For this environment, either specify a unique *container-string* for each database partition or use a relative path name.
- Although the tablespace definition is transactional and the changes to the tablespace definition are reflected in the catalog tables on commit, the buffer pool with the new definition cannot be used until the next time the

database is started. The buffer pool in use, when the ALTER TABLESPACE statement was issued, will continue to be used in the interim.

Rules:

- The BEGIN NEW STRIPE SET clause cannot be specified in the same statement as ADD, DROP, EXTEND, REDUCE, and RESIZE, unless those clauses are being directed to different partitions (SQLSTATE 429BC).
- The stripe set value specified with the TO STRIPE SET clause must be within the valid range for the tablespace being altered (SQLSTATE 42615).
- When adding or removing space from the tablespace, the following rules must be followed:
  - EXTEND and RESIZE can be used in the same statement, provided that the size of each container is increasing (SQLSTATE 429BC).
  - REDUCE and RESIZE can be used in the same statement, provided that the size of each container is decreasing (SQLSTATE 429BC).
  - EXTEND and REDUCE cannot be used in the same statement, unless they are being directed to different partitions (SQLSTATE 429BC).
  - ADD cannot be used with REDUCE or DROP in the same statement, unless they are being directed to different partitions (SQLSTATE 429BC).
  - DROP cannot be used with EXTEND or ADD in the same statement, unless they are being directed to different partitions (SQLSTATE 429BC).

Examples:

Example 1: Add a device to the PAYROLL table space.

```
ALTER TABLESPACE PAYROLL
  ADD (DEVICE '/dev/rhdisk9' 10000)
```

Example 2: Change the prefetch size and I/O overhead for the ACCOUNTING table space.

```
ALTER TABLESPACE ACCOUNTING
  PREFETCHSIZE 64
  OVERHEAD 19.3
```

Example 3: Create a tablespace TS1, then resize the containers so that all of the containers have 2000 pages (three different ALTER TABLESPACES which will accomplish this resizing are provided).

```
CREATE TABLESPACE TS1
  MANAGED BY DATABASE
  USING (FILE '/conts/cont0' 1000,
         DEVICE '/dev/rcont1' 500,
         FILE 'cont2' 700)
```

```
ALTER TABLESPACE TS1
  RESIZE (FILE '/conts/cont0' 2000,
            DEVICE '/dev/rcont1' 2000,
            FILE 'cont2' 2000)
```

OR

```
ALTER TABLESPACE TS1
  RESIZE (ALL 2000)
```

OR

```
ALTER TABLESPACE TS1
  EXTEND (FILE '/conts/cont0' 1000,
            DEVICE '/dev/rcont1' 1500,
            FILE 'cont2' 1300)
```

Example 4: Extend all of the containers in the DATA_TS tablespace by 1000 pages.

```
ALTER TABLESPACE DATA_TS
  EXTEND (ALL 1000)
```

Example 5: Resize all of the containers in the INDEX_TS tablespace to 100 megabytes (MB).

```
ALTER TABLESPACE INDEX_TS
  RESIZE (ALL 100 M)
```

Example 6: Add three new containers. Extend the first container, and resize the second.

```
ALTER TABLESPACE TS0
  ADD (FILE 'cont2' 2000, FILE 'cont3' 2000)
  ADD (FILE 'cont4' 2000)
  EXTEND (FILE 'cont0' 100)
  RESIZE (FILE 'cont1' 3000)
```

Example 7: Table space TSO exists on partitions 0, 1 and 2. Add a new container to database partition 0. Extend all of the containers on database partition 1. Resize a container on all database partitions other than the ones that were explicitly specified (that is, database partitions 0 and 1).

```
ALTER TABLESPACE TS0
  ADD (FILE 'A' 200) ON DBPARTITIONNUM (0)
  EXTEND (ALL 200) ON DBPARTITIONNUM (1)
  RESIZE (FILE 'B' 500)
```

The RESIZE clause is the default container clause in this example, and will be executed on database partition 2, because other operations are being explicitly sent to database partitions 0 and 1. If, however, there had only been these two database partitions, the statement would have succeeded, but returned a warning (SQL1758W) that default containers had been specified but not used.

Related reference:

- "CREATE TABLESPACE" on page 395

ALTER TYPE (Structured)

The ALTER TYPE statement is used to add or drop attributes or method specifications of a user-defined structured type. Properties of existing methods can also be altered.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- ALTERIN privilege on the schema of the type
- Definer of the type, as recorded in the DEFINER column of SYSCAT.DATATYPES

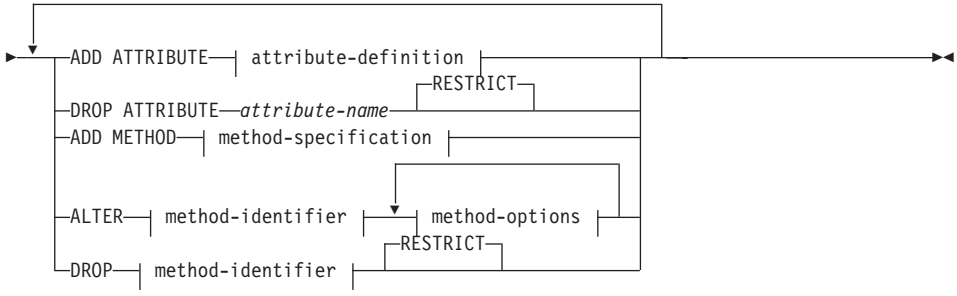
To alter a method to be not fenced, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SYSADM or DBADM authority
- CREATE_NOT_FENCED_ROUTINE authority on the database

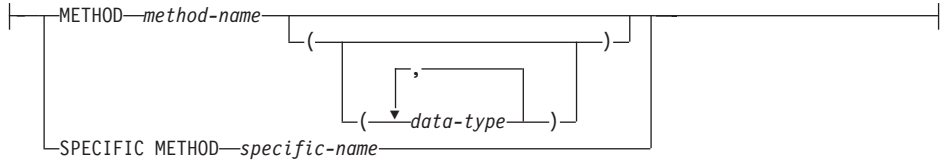
To alter a method to be fenced, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:



method-identifier:



method-options:

Description:

type-name
Identifies the structured type to be changed. It must be an existing type defined in the catalog (SQLSTATE 42704), and the type must be a structured type (SQLSTATE 428DP). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

ADD ATTRIBUTE
Adds an attribute after the last attribute of the existing structured type.

attribute-definition
Defines the attributes of the structured type.

attribute-name
Specifies a name for the attribute. The name cannot be the same as any other attribute of this structured type (including inherited attributes) or any subtype of this structured type (SQLSTATE 42711).

A number of names used as keywords in predicates are reserved for system use, and may not be used as an *attribute-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators.

data-type 1
Specifies the data type of the attribute. It is one of the data types listed under CREATE TABLE, other than LONG VARCHAR, LONG VARGRAPHIC, or a distinct type based on LONG VARCHAR or LONG VARGRAPHIC (SQLSTATE 42601). The data type must identify an existing data type (SQLSTATE 42704). If *data-type* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The description of various data types is given in "CREATE TABLE". If the attribute data type is a reference type, the target type of the reference must be a structured type that exists (SQLSTATE 42704).

A structured type defined with an attribute of type DATALINK can only be effectively used as the data type for a typed table or a typed view (SQLSTATE 01641).

To prevent type definitions that, at run time, would permit an instance of the type to directly, or indirectly, contain another instance of the same type or one of its subtypes, there is a restriction that a type may not be defined such that one of its attribute types directly or indirectly uses itself (SQLSTATE 428EP).

lob-options
Specifies the options associated with LOB types (or distinct types based on LOB types). For a detailed description of lob-options, see "CREATE TABLE".

datalink-options
Specifies the options associated with DATALINK types (or distinct types based on DATALINK types). For a detailed descriptions of datalink-options, see "CREATE TABLE".

Note that if no options are specified for a DATALINK type, or distinct type sourced on the DATALINK type, LINKTYPE URL and NO LINK CONTROL are the default options.

DROP ATTRIBUTE
Drops an attribute of the existing structured type.

attribute-name
The name of the attribute. The attribute must exist as an attribute of the type (SQLSTATE 42703).

RESTRICT
Enforces the rule that no attribute can be dropped if *type-name* is used as the type of an existing table, view, column, attribute nested inside the type of a column, or an index extension.

ADD METHOD *method-specification*
Adds a method specification to the type identified by *type-name*. The method cannot be used until a separate CREATE METHOD statement is used to give the method a body. For more information about *method-specification*, see "CREATE TYPE (Structured)".

ALTER *method-identifier*
Uniquely identifies an instance of a method that is to be altered. The specified method may or may not have an existing method body. Methods declared as LANGUAGE SQL cannot be altered (SQLSTATE 42917).

method-identifier

METHOD *method-name*
Identifies a particular method, and is valid only if there is exactly one method instance with the name *method-name* for the type *type-name*. The identified method can have any number of parameters defined for it. If no method by this name exists for the type, an error (SQLSTATE 42704) is raised. If there is more than one instance of the method for the type, an error (SQLSTATE 42725) is raised.

METHOD *method-name (data-type,...)*
Provides the method signature, which uniquely identifies the method. The method resolution algorithm is not used.

method-name
Specifies the name of the method for the type *type-name*.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE TYPE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific method instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE TYPE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no method with the specified signature exists for the type in the named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC METHOD *specific-name*
Identifies a particular method, using the name that is specified or defaulted to at method creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific method instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

method-options
Specifies the options that are to be altered for the method.

FENCED or NOT FENCED
Specifies whether the method is considered safe to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED). Most methods have the option of running as FENCED or NOT FENCED.

If a method is altered to be FENCED, the database manager insulates its internal resources (for example, data buffers) from access by the method. In general, a method running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION:
Use of NOT FENCED for methods that were not adequately coded, reviewed, and tested can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED methods are used.

A method declared as NOT THREADSAFE cannot be altered to be NOT FENCED (SQLSTATE 42613).

If a method has any parameters defined AS LOCATOR, and was defined with the NO SQL option, the method cannot be altered to be FENCED (SQLSTATE 42613).

This option cannot be altered for LANGUAGE OLE methods (SQLSTATE 42849).

THREADSAFE or NOT THREADSAFE
Specifies whether a method is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the method is defined with LANGUAGE other than OLE:

- If the method is defined as THREADSAFE, the database manager can invoke the method in the same process as other routines. In general, to be threadsafe, a method should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED methods can be THREADSAFE. If the method is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).
- If the method is defined as NOT THREADSAFE, the database manager will never invoke the method in the same process as another routine. Only a fenced method can be NOT THREADSAFE (SQLSTATE 42613).

DROP *method-identifier*
Uniquely identifies an instance of a method that is to be dropped. The specified method must not have an existing method body (SQLSTATE 428ER). Use the DROP METHOD statement to drop the method body before using ALTER TYPE DROP METHOD. Methods implicitly generated by the CREATE TYPE statement (such as mutators and observers) cannot be dropped (SQLSTATE 42917).

RESTRICT
Indicates that the specified method is restricted from having an existing method body. Use the DROP METHOD statement to drop the method body before using ALTER TYPE DROP METHOD.

Rules:

- Adding or dropping an attribute is not allowed for type *type-name* (SQLSTATE 55043) if either:
  - The type or one of its subtypes is the type of an existing table or view.
  - There exists a column of a table whose type directly or indirectly uses *type-name*. The terms *directly uses* and *indirectly uses* are defined in "Structured types".

  - The type or one of its subtypes is used in an index extension.
- A type may not be altered by adding attributes so that the total number of attributes for the type, or any of its subtypes, exceeds 4082 (SQLSTATE 54050).
- ADD ATTRIBUTE option:
  - ADD ATTRIBUTE generates observer and mutator methods for the new attribute. These methods are similar to those generated when a structured type is created (see "CREATE TYPE (Structured)"). If these methods conflict with or override any existing methods or functions, the ALTER TYPE statement fails (SQLSTATE 42745).
  - If the INLINE LENGTH for the type (or any of its subtypes) was explicitly specified by the user with a value less than 292, and the attributes added cause the specified inline length to be less than the size of the result of the constructor function for the altered type (32 bytes plus 10 bytes per attribute), then an error results (SQLSTATE 42611).
- DROP ATTRIBUTE option:
  - An attribute that is inherited from an existing supertype cannot be dropped (SQLSTATE 428DJ).
  - DROP ATTRIBUTE drops the mutator and observer methods of the dropped attributes, and checks dependencies on those dropped methods.
- DROP METHOD option:
  - An original method that is overridden by other methods cannot be dropped (SQLSTATE -2).

Notes:

- It is not possible to alter a method that is in the SYSIBM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).
- When a type is altered by adding or dropping an attribute, all packages are invalidated that depend on functions or methods that use this type or a subtype of this type as a parameter or a result.
- When an attribute is added to or dropped from a structured type:
  - If the INLINE LENGTH of the type was calculated by the system when the type was created, the INLINE LENGTH values are automatically modified for the altered type, and all of its subtypes to account for the change. The INLINE LENGTH values are also automatically (recursively) modified for all structured types where the INLINE LENGTH was calculated by the system and the type includes an attribute of any type with a changed INLINE LENGTH.
  - If the INLINE LENGTH of any type affected by adding or dropping attributes was explicitly specified by a user, then the INLINE LENGTH for that particular type is not changed. Special care must be taken for explicitly specified inline lengths. If it is likely that a type will have

attributes added later on, then the inline length, for any uses of that type or one of its subtypes in a column definition, should be large enough to account for the possible increase in length of the instantiated object.

– If new attributes are to be made visible to application programs, existing transform functions must be modified to match the new structure of the data type.

- *Privileges*

  The EXECUTE privilege is not given for any methods explicitly specified in the ALTER TYPE statement until a method body is defined using the CREATE METHOD statement. The definer of the user-defined type has the ability to drop the method specification using the ALTER TYPE statement.

Examples:

Example 1: The ALTER TYPE statement can be used to permit a cycle of mutually referencing types and tables. Consider mutually referencing tables named EMPLOYEE and DEPARTMENT.

The following sequence would allow the types and tables to be created.

```
CREATE TYPE DEPT ...
CREATE TYPE EMP ... (including attribute named DEPTREF of type REF(DEPT))
ALTER TYPE DEPT ADD ATTRIBUTE MANAGER REF(EMP)
CREATE TABLE DEPARTMENT OF DEPT ...
CREATE TABLE EMPLOYEE OF EMP (DEPTREF WITH OPTIONS SCOPE DEPARTMENT)
ALTER TABLE DEPARTMENT ALTER COLUMN MANAGER ADD SCOPE EMPLOYEE
```

The following sequence would allow these tables and types to be dropped.

```
DROP TABLE EMPLOYEE   (the MANAGER column in DEPARTMENT becomes unscoped)
DROP TABLE DEPARTMENT
ALTER TYPE DEPT DROP ATTRIBUTE MANAGER
DROP TYPE EMP
DROP TYPE DEPT
```

Example 2: The ALTER TYPE statement can be used to create a type with an attribute that references a subtype.

```
CREATE TYPE EMP ...
CREATE TYPE MGR UNDER EMP ...
ALTER TYPE EMP ADD ATTRIBUTE MANAGER REF(MGR)
```

Example 3: The ALTER TYPE statement can be used to add an attribute. The following statement adds the SPECIAL attribute to the EMP type. Because the inline length was not specified on the original CREATE TYPE statement, DB2 recalculates the inline length by adding 13 (10 bytes for the new attribute + attribute length + 2 bytes for a non-LOB attribute).

```
ALTER TYPE EMP ...
  ADD ATTRIBUTE SPECIAL CHAR(1)
```

Example 4: The ALTER TYPE statement can be used to add a method associated with a type. The following statement adds a method called BONUS.

```
ALTER TYPE EMP ...
  ADD METHOD BONUS (RATE DOUBLE)
    RETURNS INTEGER
    LANGUAGE SQL
    CONTAINS SQL
    NO EXTERNAL ACTION
    DETERMINISTIC
```

Note that the BONUS method cannot be used until a CREATE METHOD statement is issued to create the method body. If it is assumed that type EMP includes an attribute called SALARY, then the following is an example of a method body definition.

```
CREATE METHOD BONUS(RATE DOUBLE) FOR EMP
  RETURN CAST(SELF.SALARY * RATE AS INTEGER)
```

Related reference:

- "CREATE TABLE" on page 332
- "CREATE TYPE (Structured)" on page 427
- "CREATE METHOD" on page 285
- "User-defined types" in the *SQL Reference, Volume 1*

Related samples:

- "dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)"

ALTER USER MAPPING

The ALTER USER MAPPING statement is used to change the authorization ID or password that is used at a data source for a specified federated server authorization ID.

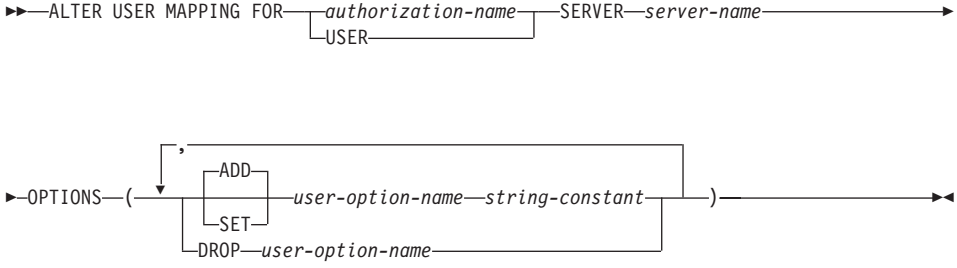
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

If the authorization ID of the statement is different than the authorization name that is mapped to the data source, then the authorization ID of the statement must include SYSADM or DBADM authority. Otherwise, if the authorization ID and the authorization name match, then no privileges or authorities are required.

Syntax:



Description:

authorization-name
Specifies the authorization name under which a user or application connects to a federated database.

USER
The value in the special register USER. When USER is specified, then the authorization ID of the ALTER USER MAPPING statement will be mapped to the data source authorization ID that is specified in the REMOTE_AUTHID user option.

SERVER *server-name*
Identifies the data source accessible under the remote authorization ID

that maps to the local authorization ID that's denoted by *authorization-name* or referenced by USER.

OPTIONS
Indicates what user options are to be enabled, reset, or dropped for the mapping that is being altered.

ADD
Enables a user option.

SET
Changes the setting of a user option.

user-option-name
Names a user option that is to be enabled or reset.

string-constant
Specifies the setting for *user-option-name* as a character string constant.

DROP *user-option-name*
Drops a user option.

Notes:

- A user option cannot be specified more than once in the same ALTER USER MAPPING statement (SQLSTATE 42853). When a user option is enabled, reset, or dropped, any other user options that are in use are not affected.
- A user mapping cannot be altered in a given unit of work (UOW) if the UOW already includes a SELECT statement that references a nickname for a table or view at the data source that is to be included in the mapping.

Examples:

Example 1: Jim uses a local database to connect to an Oracle data source called ORACLE1. He accesses the local database under the authorization ID KLEEWEIN; KLEEWEIN maps to CORONA, the authorization ID under which he accesses ORACLE1. Jim is going to start accessing ORACLE1 under a new ID, JIMK. So KLEEWEIN now needs to map to JIMK.

```
ALTER USER MAPPING FOR KLEEWEIN
  SERVER ORACLE1
  OPTIONS ( SET REMOTE_AUTHID 'JIMK'  )
```

Example 2: Mary uses a federated database to connect to a DB2 Universal Database for OS/390 and z/OS data source called DORADO. She uses one authorization ID to access DB2 and another to access DORADO, and she has created a mapping between these two IDs. She has been using the same password with both IDs, but now decides to use a separate password, ZNYQ, with the ID for DORADO. Accordingly, she needs to map her federated database password to ZNYQ.

```
ALTER USER MAPPING FOR MARY
  SERVER DORADO
  OPTIONS ( ADD REMOTE_PASSWORD 'ZNYQ'  )
```

Related reference:

- "User options for federated systems" in the *Federated Systems Guide*

ALTER VIEW

The ALTER VIEW statement modifies an existing view by altering a reference type column to add a scope.

Invocation:

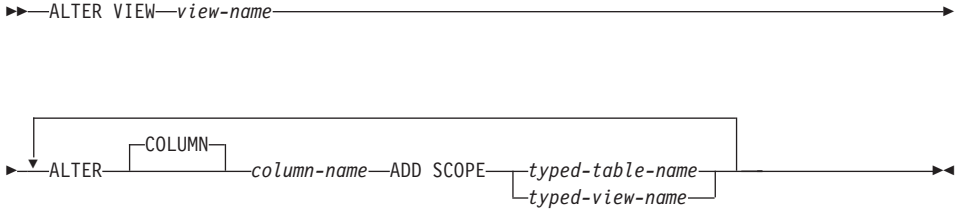
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- ALTERIN privilege on the schema of the view
- Definer of the view to be altered
- CONTROL privilege on the view to be altered.

Syntax:



Description:

view-name
Identifies the view to be changed. It must be a view described in the catalog.

ALTER COLUMN *column-name*
Is the name of the column to be altered in the view. The *column-name* must identify an existing column of the view (SQLSTATE 42703). The name cannot be qualified.

ADD SCOPE
Add a scope to an existing reference type column that does not already have a scope defined (SQLSTATE 428DK). The column must not be inherited from a superview (SQLSTATE 428DJ).

typed-table-name
The name of a typed table. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

typed-view-name
The name of a typed view. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

ALTER WRAPPER

The ALTER WRAPPER statement is used to update the properties of a wrapper.

Invocation:

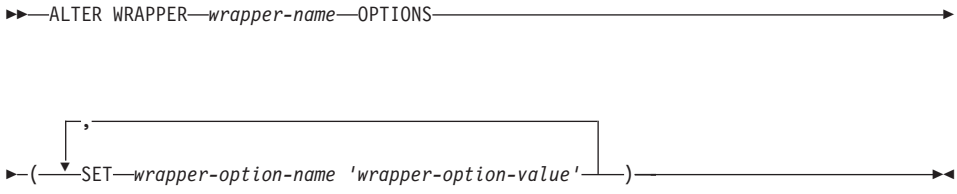
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority

Syntax:



Description:

wrapper-name
Specifies the name of the wrapper.

OPTIONS (SET *wrapper-option-name 'wrapper-option-value', ...***)**
Currently, the only supported wrapper option name is DB2_FENCED; valid values for this option are 'Y' or 'N'.

Example:

Example 1: Set the DB2_FENCED option on for wrapper SQLNET.

```
ALTER WRAPPER SQLNET OPTIONS (SET DB2_FENCED 'Y')
```

Related reference:

- "CREATE WRAPPER" on page 479

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION statement marks the beginning of a host variable 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 REXX.

Authorization:

None required.

Syntax:

Description:
The BEGIN DECLARE SECTION statement may be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. It is used to indicate the beginning of a host variable declaration section. A host variable section ends with an END DECLARE SECTION statement.
Rules:

- The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.
- SQL statements cannot be included within the declare section.
- Variables referenced in SQL statements must be declared in a declare section in all host languages other than REXX. Furthermore, the section must appear before the first reference to the variable. Generally, host variables are not declared in REXX with the exception of LOB locators and file reference variables. In this case, they are not declared within a BEGIN DECLARE SECTION.
- Variables declared outside a declare section should not have the same name as variables declared within a declare section.
- LOB data types must have their data type and length preceded with the SQL TYPE IS keywords.

Examples:

Example 1: Define the host variables hv_smint (smallint), hv_vchar24 (varchar(24)), hv_double (double), hv_blob_50k (blob(51200)), hv_struct (of structured type "struct_type" as blob(10240)) in a C program.

```
EXEC SQL  BEGIN DECLARE SECTION;
  short  hv_smint;
  struct {
  short hv_vchar24_len;
  char  hv_vchar24_value[24];
 }                              hv_vchar24;
  double hv_double;
  SQL TYPE IS BLOB(50K) hv_blob_50k;
  SQL TYPE IS struct_type AS BLOB(10k) hv_struct;
EXEC SQL  END DECLARE SECTION;
```

Example 2: Define the host variables HV-SMINT (smallint), HV-VCHAR24 (varchar(24)), HV-DEC72 (dec(7,2)), and HV-BLOB-50k (blob(51200)) in a COBOL program.

```
WORKING-STORAGE SECTION.
      EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
01  HV-SMINT              PIC S9(4)       COMP-4.
01  HV-VCHAR24.
    49 HV-VCHAR24-LENGTH  PIC S9(4)       COMP-4.
    49 HV-VCHAR24-VALUE   PIC X(24).
01  HV-DEC72              PIC S9(5)V9(2)  COMP-3.
01 HV-BLOB-50K            USAGE SQL TYPE IS BLOB(50K).
      EXEC SQL END DECLARE SECTION  END-EXEC.
```

Example 3: Define the host variables HVSMINT (smallint), HVVCHAR24 (char(24)), HVDOUBLE (double), and HVBLOB50k (blob(51200)) in a Fortran program.

```
EXEC SQL  BEGIN DECLARE SECTION
  INTEGER*2      HVSMINT
  CHARACTER*24   HVVCHAR24
  REAL*8         HVDOUBLE
  SQL TYPE IS BLOB(50K)  HVBLOB50K
EXEC SQL  END DECLARE SECTION
```

Note: In Fortran, if the expected value is greater than 254 characters, then a CLOB host variable should be used.

Example 4: Define the host variables HVSMINT (smallint), HVBLOB50K (blob(51200)), and HVCLOBLOC (a CLOB locator) in a REXX program.

```
DECLARE :HVCLOBLOC  LANGUAGE TYPE CLOB LOCATOR
call sqlexec 'FETCH c1 INTO :HVSMINT, :HVBLOB50K'
```

Note that the variables HVSMINT and HVBLOB50K were implicitly defined by using them in the FETCH statement.

Related reference:

- "END DECLARE SECTION" on page 542

Related samples:

- "advsql.sqb -- How to read table data using CASE (MF COBOL)"
- "dtlob.sqc -- How to use the LOB data type (C)"
- "spclient.sqc -- Call various stored procedures (C)"
- "tut_read.sqc -- How to read tables (C)"
- "udfemsrv.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "spclient.sqC -- Call various stored procedures (C++)"
- "tut_read.sqC -- How to read tables (C++)"
- "udfemsrv.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)"

CALL

The CALL statement calls a procedure.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

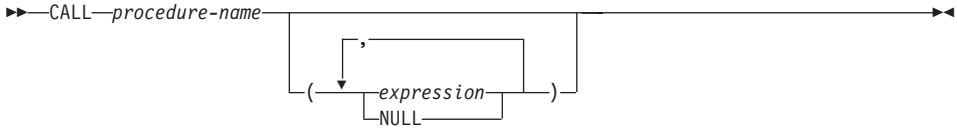
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- EXECUTE privilege on the procedure
- SYSADM or DBADM authority

If a matching procedure exists that the authorization ID of the statement is not authorized to execute, an error (SQLSTATE 42501) is returned.

Syntax:



Description:

procedure-name
Specifies the procedure that is to be called. It must be a procedure that is described in the catalog. The specific procedure to invoke is chosen using procedure resolution. (For more details, see the "Notes" section of this statement.)

expression **or NULL**
Each specification of *expression* or NULL is an argument of the CALL. The *n*th argument of the CALL statement corresponds to the *n*th parameter defined in the CREATE PROCEDURE statement for the procedure.

Each argument of the CALL must be compatible with the corresponding parameter in the procedure definition as follows:

- IN parameter
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the storage assignment rules.

- OUT parameter
  - The argument must be a single host variable or parameter marker (SQLSTATE 42886).
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the retrieval assignment rules.
- INOUT parameter
  - The argument must be a single host variable or parameter marker (SQLSTATE 42886).
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the storage assignment rules on invocation and the retrieval assignment rules on return.

Notes:

- ***Procedure signatures:***

  A procedure is identified by its schema, a procedure name, and the number of parameters. This is called a procedure signature, which must be unique within the database. There can be more than one procedure with the same name in a schema, provided that the number of parameters is different for each procedure.
- ***SQL path:***

  A procedure can be invoked by referring to a qualified name (schema and procedure name), followed by an optional list of arguments enclosed by parentheses. A procedure can also be invoked without the schema name, resulting in a choice of possible procedures in different schemas with the same number of parameters. In this case, the SQL path is used to assist in procedure resolution. The SQL path is a list of schemas that is searched to identify a procedure with the same name and number of parameters. For static CALL statements, SQL path is specified using the FUNCPATH bind option. For dynamic CALL statements, SQL path is the value of the CURRENT PATH special register.
- ***Procedure resolution:***

  Given a procedure invocation, the database manager must decide which of the possible procedures with the same name to call. Procedure resolution is done using the steps that follow.

  1. Find all procedures from the catalog (SYSCAT.ROUTINES), such that all of the following are true:
     - For invocations where the schema name was specified (that is, qualified references), the schema name and the procedure name match the invocation name.

- For invocations where the schema name was not specified (that is, unqualified references), the procedure name matches the invocation name, and has a schema name that matches one of the schemas in the SQL path.
- The number of defined parameters matches the invocation.
- The invoker has the EXECUTE privilege on the procedure.

2. Choose the procedure whose schema is earliest in the SQL path.

If there are no candidate procedures remaining after step 1, an error is returned (SQLSTATE 42884).

- ***Retrieving the RETURN_STATUS from an SQL procedure:***

  If an SQL procedure successfully issues a RETURN statement with a status value, this value is returned in the first SQLERRD field of the SQLCA. If the CALL statement is issued in an SQL procedure, use the GET DIAGNOSTICS statement to retrieve the RETURN_STATUS value. The value is –1 if the SQLSTATE indicates an error. The values is 0 if no error is returned and the RETURN statement was not specified in the procedure.

- ***Returning result sets from procedures:***

  If the calling program is written using CLI, JDBC, or SQLj, or the caller is an SQL procedure, result sets can be returned directly to the caller. The procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on the result set, and leaving the cursor open when exiting the procedure.

  At the end of a procedure:

  - For every cursor that has been left open, a result set is returned to the caller or (for WITH RETURN TO CLIENT cursors) directly to the client.
  - Only unread rows are passed back. For example, if the 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, rows 151 through 500 will be returned to the caller or application (as appropriate).

  If the procedure was invoked from CLI or JDBC, and more than one cursor is left open, the result sets can only be processed in the order in which the cursors were opened.

- ***Improving performance:***

  The values of all arguments are passed from the application to the procedure. To improve the performance of this operation, host variables that correspond to OUT parameters and have lengths of more than a few bytes should be set to NULL before the CALL statement is executed.

- ***Nesting CALL statements:***

  Procedures can be called from routines as well as application programs. When a procedure is called from a routine, the call is considered to be nested.

If a procedure returns any query result sets, the result sets are returned as follows:

– RETURN TO CALLER result sets are visible only to the program that is at the previous nesting level.
– RETURN TO CLIENT results sets are visible only if the procedure was invoked from a set of nested procedures. If a function or method occurs anywhere in the call chain, the result set is not visible. If the result set is visible, it is only visible to the client application that made the initial procedure call.

Consider the following example:

```
Client program:
EXEC SQL CALL PROCA;

  PROCA:
  EXEC SQL CALL PROCB;

    PROCB:
    EXEC SQL DECLARE B1 CURSOR WITH RETURN TO CLIENT ...;
    EXEC SQL DECLARE B2 CURSOR WITH RETURN TO CALLER ...;
    EXEC SQL DECLARE B3 CURSOR FOR SELECT UDFA FROM T1;

      UDFA:
      EXEC SQL CALL PROCC;

        PROCC:
        EXEC SQL DECLARE C1 CURSOR WITH RETURN TO CLIENT ...;
        EXEC SQL DECLARE C2 CURSOR WITH RETURN TO CALLER ...;
```

From procedure PROCB:

– Cursor B1 is visible in the client application, but not visible in procedure PROCA.
– Cursor B2 is visible in PROCA, but not visible to the client.

From procedure PROCC:

– Cursor C1 is visible to neither UDFA nor to the client application. (Because UDFA appears in the call chain between the client and PROCC, the result set is not returned to the client.)
– Cursor C2 is visible in UDFA, but not visible to any of the higher procedures.

• ***Nesting procedures within functions or methods:***

When a procedure is called from a function or method, there are additional restrictions on the statements that may be issued from the procedure. Specifically:

– The procedure may not issue COMMIT or ROLLBACK unit of work statements.

  - The procedure has the same table access restrictions as the invoking function or method.
  - RETURN TO CLIENT result sets will not be visible to the client.
  - Savepoints defined before the function or method was invoked will not be visible to the procedure, and savepoints defined inside the procedure will not be visible outside the function or method.
- ***Compatibilities:***
  - There is an older form of the CALL statement that can be embedded in applications by precompiling the application with the CALL_RESOLUTION DEFERRED option. This option is not available for SQL procedures.

Examples:

Example 1:

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)
                EXTERNAL NAME 'parts!onhand'
                LANGUAGE JAVA
                PARAMETER STYLE DB2GENERAL;
```

A Java application calls this procedure using the following code fragment:

```
...
CallableStatement stpCall;

String sql = "CALL PARTS_ON_HAND (?, ?, ?)";

stpCall = con.prepareCall(sql); /*con is the connection */

stpCall.setInt(1, hvPartnum);
stpCall.setBigDecimal(2, hvCost);
stpCall.setInt(3, hvQuantity);

stpCall.registerOutParameter(2, Types.DECIMAL, 2);
stpCall.registerOutParameter(3, Types.INTEGER);

stpCall.execute();

hvCost = stpCall.getBigDecimal(2);
hvQuantity = stpCall.getInt(3);
...
```

This application code fragment will invoke the Java method `onhand` in class `parts`, because the procedure name specified on the CALL statement is found in the database and has the external name `parts!onhand`.

Example 2:

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

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

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

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

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

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

Following through the algorithm...

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

Related reference:

- “CURRENT PATH special register” in the *SQL Reference, Volume 1*
- “CALL invoked from a compiled statement” in the *SQL Reference, Volume 1*
- “Assignments and comparisons” in the *SQL Reference, Volume 1*

Related samples:

- “outcli.sqb -- Call stored procedures using the SQLDA structure (MF COBOL)”
- “spclient.c -- Call various stored procedures (CLI)”
- “spclient.sqc -- Call various stored procedures (C)”
- “spclient.sqC -- Call various stored procedures (C++)”
- “SpClient.sqlj -- Call a variety of types of stored procedures from SpServer.sqlj (SQLj)”

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 be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared.

Authorization:

None required. For the authorization required to use a cursor, see "DECLARE 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.

WITH RELEASE
The release of all read locks that have been held for the cursor is attempted. Note that not all of the read locks are necessarily released; these locks may be held for other operations or activities.

Notes:

- At the end of a unit of work, all cursors that belong to an application process and that were declared without the WITH HOLD option are implicitly closed.
- The WITH RELEASE clause has no effect when closing cursors defined in functions or methods. The clause also has no effect when closing cursors defined in stored procedures called from functions or methods.
- The WITH RELEASE clause has no effect for cursors that are operating under isolation levels CS or UR. When specified for cursors that are operating under isolation levels RS or RR, WITH RELEASE terminates some of the guarantees of those isolation levels. Specifically, if the cursor is

opened again, an RS cursor may experience the 'nonrepeatable read' phenomenon and an RR cursor may experience either the 'nonrepeatable read' or 'phantom' phenomenon.

If a cursor that was originally either RR or RS is reopened after being closed using the WITH RELEASE clause, new read locks will be acquired.

- Special rules apply to cursors within a stored procedure that have not been closed before returning to the calling program.
- While a cursor is open (that is, it has not been closed yet), any changes to sequence values as a result of statements involving that cursor (for example, a FETCH or an UPDATE using the cursor that includes a NEXTVAL expression for a sequence) will not result in an update to PREVVAL for those sequences as seen by that cursor. The PREVVAL values for these affected sequences are updated when the cursor is closed.

Example:

A cursor is used to fetch one row at a time into the C program variables `dnum`, `dname`, and `mnum`. Finally, the cursor is closed. If the cursor is reopened, it is again located at the beginning of the rows to be fetched.

```
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;
```

Related reference:

- "CALL" on page 101
- "DECLARE CURSOR" on page 482
- "Comparison of isolation levels" in the *SQL Reference, Volume 1*

Related samples:

- "dynamic.sqb -- How to update table data with cursor dynamically (MF COBOL)"
- "tut_mod.sqc -- How to modify table data (C)"
- "tut_read.sqc -- How to read tables (C)"
- "tut_mod.sqC -- How to modify table data (C++)"
- "tut_read.sqC -- How to read tables (C++)"

COMMENT

The COMMENT statement adds or replaces comments in the catalog descriptions of various objects.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

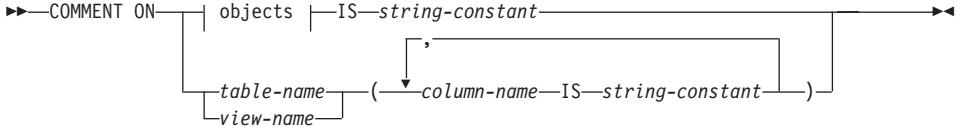
Authorization:

The privileges that must be held by the authorization ID of the COMMENT statement must include one of the following:

- SYSADM or DBADM
- definer of the object (underlying table for column or constraint) as recorded in the DEFINER column of the catalog view for the object (OWNER column for a schema)
- ALTERIN privilege on the schema (applicable only to objects allowing more than one-part names)
- CONTROL privilege on the object (applicable to index, package, table and view objects only)
- ALTER privilege on the object (applicable to table objects only)

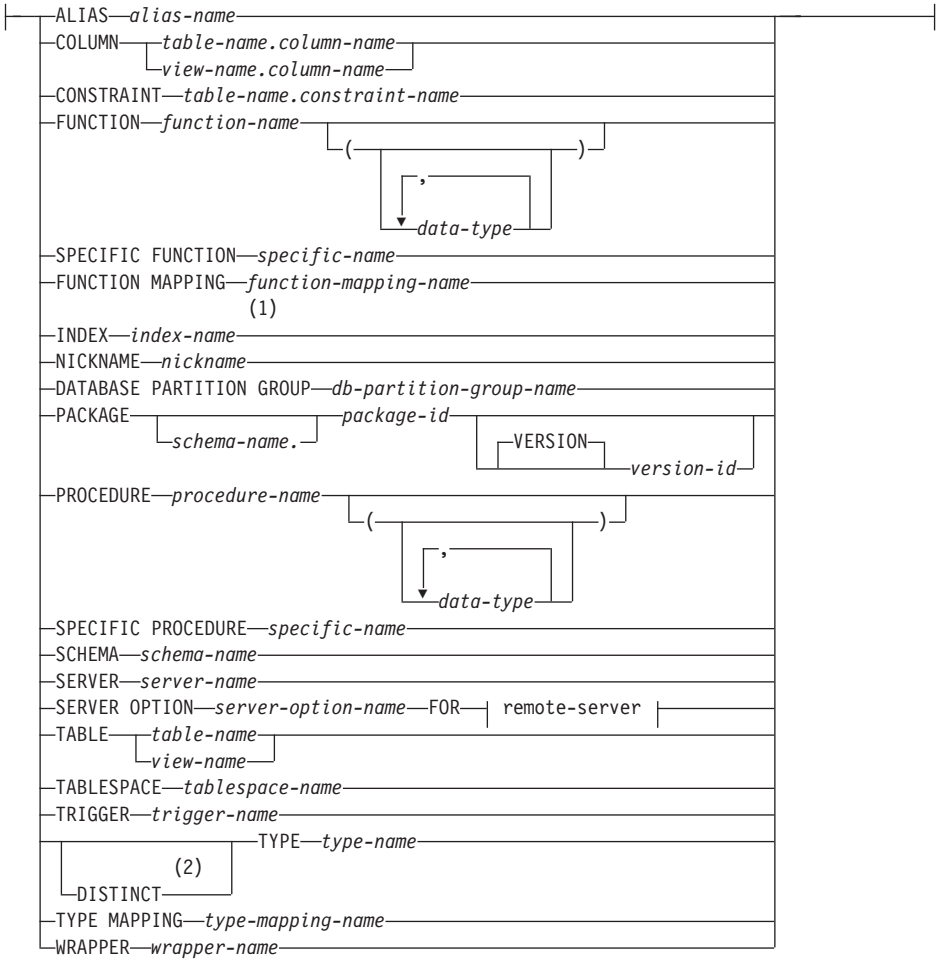
Note that for table space or database partition group, the authorization ID must have SYSADM or SYSCTRL authority.

Syntax:

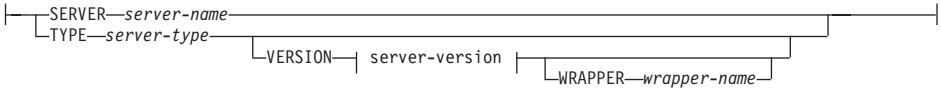


objects:

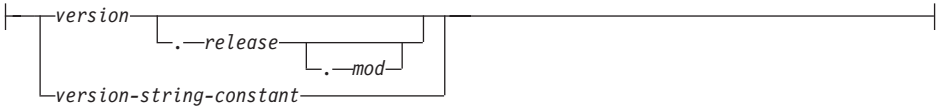
COMMENT



remote-server:



server-version:



Notes:

1 *Index-name* can be the name of either an index or an index specification.

2 The keyword DATA can be used as a synonym for DISTINCT.

Description:

ALIAS *alias-name*
Indicates a comment will be added or replaced for an alias. The *alias-name* must identify an alias that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the alias.

COLUMN *table-name.column-name* or *view-name.column-name*
Indicates a comment will be added or replaced for a column. The *table-name.column-name* or *view-name.column-name* combination must identify a column and table combination that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.COLUMNS catalog view for the row that describes the column.

A comment cannot be made on a column of an inoperative view. (SQLSTATE 51024).

CONSTRAINT *table-name.constraint-name*
Indicates a comment will be added or replaced for a constraint. The *table-name.constraint-name* combination must identify a constraint and the table that it constrains; they must be described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.TABCONST catalog view for the row that describes the constraint.

FUNCTION
Indicates a comment will be added or replaced for a function. The function instance specified must be a user-defined function or function template described in the catalog.

There are several different ways available to identify the function instance:

FUNCTION *function-name*
Identifies the particular function, and is valid only if there is exactly one function with the *function-name*. The function thus identified may have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the function in the named or implied schema, an error (SQLSTATE 42725) is raised.

FUNCTION *function-name* **(***data-type***,...)**
Provides the function signature, which uniquely identifies the function to be commented upon. The function selection algorithm is *not* used.

function-name
Gives the function name of the function to be commented upon. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(*data-type***,...)**
Must match the data types that were specified on the CREATE FUNCTION statement in the corresponding position. The number of data types, and the logical concatenation of the data types is used to identify the specific function for which to add or replace the comment.

If the *data-type* is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(n) does not need to match the defined value for n since 0 <n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

(Note that the FOR BIT DATA attribute is not considered part of the signature for matching purposes. So, for example, a CHAR FOR BIT DATA specified in the signature would match a function defined with CHAR only, and vice versa.)

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC FUNCTION *specific-name*
Indicates that comments will be added or replaced for a function (see FUNCTION for other methods of identifying a function). Identifies the particular user-defined function that is to be commented upon, using

the specific name either specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

It is not possible to comment on a function that is in the SYSIBM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).

The comment replaces the value of the REMARKS column of the SYSCAT.ROUTINES catalog view for the row that describes the function.

FUNCTION MAPPING *function-mapping-name*
Indicates a comment will be added or replaced for a function mapping. The *function-mapping-name* must identify a function mapping that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.FUNCMAPPINGS catalog view for the row that describes the function mapping.

INDEX *index-name*
Indicates a comment will be added or replaced for an index or index specification. The *index-name* must identify either a distinct index or an index specification that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.INDEXES catalog view for the row that describes the index or index specification.

NICKNAME *nickname*
Indicates a comment will be added or replaced for a nickname. The *nickname* must be a nickname that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the nickname.

DATABASE PARTITION GROUP *db-partition-group-name*
Indicates a comment will be added or replaced for a database partition group. The *db-partition-group-name* must identify a distinct database partition group that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.DBPARTITIONGROUPS catalog view for the row that describes the database partition group.

PACKAGE *schema-name.package-id*
Indicates that a comment will be added or replaced for a package. If a schema name is not specified, the package ID is implicitly qualified by the default schema. The schema name and package ID, together with the implicitly or explicitly specified version ID, must identify a package that

is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.PACKAGES catalog view for the row that describes the package.

VERSION *version-id*
Identifies which package version is to be commented on. If a value is not specified, the version defaults to the empty string. If multiple packages with the same package name but different versions exist, only one package version can be commented on in one invocation of the COMMENT statement.

PROCEDURE
Indicates a comment will be added or replaced for a procedure. The procedure instance specified must be a stored procedure described in the catalog.

There are several different ways available to identify the procedure instance:

PROCEDURE *procedure-name*
Identifies the particular procedure, and is valid only if there is exactly one procedure with the *procedure-name* in the schema. The procedure thus identified may have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the procedure in the named or implied schema, an error (SQLSTATE 42725) is raised.

PROCEDURE *procedure-name* **(***data-type***,...)**
This is used to provide the procedure signature, which uniquely identifies the procedure to be commented upon.

procedure-name
Gives the procedure name of the procedure to be commented upon. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(*data-type***,...)**
Must match the data types that were specified on the CREATE PROCEDURE statement in the corresponding position. The

number of data types, and the logical concatenation of the data types is used to identify the specific procedure for which to add or replace the comment.

If the *data-type* is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE PROCEDURE statement.

A type of FLOAT(n) does not need to match the defined value for n since 0<n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC PROCEDURE *specific-name*
Indicates that comments will be added or replaced for a procedure (see PROCEDURE for other methods of identifying a procedure). Identifies the particular stored procedure that is to be commented upon, using the specific name either specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

It is not possible to comment on a procedure that is in the SYSIBM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).

The comment replaces the value of the REMARKS column of the SYSCAT.ROUTINES catalog view for the row that describes the procedure.

SCHEMA *schema-name*
Indicates a comment will be added or replaced for a schema. The *schema-name* must identify a schema that is described in the catalog

(SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.SCHEMATA catalog view for the row that describes the schema.

SERVER *server-name*
Indicates a comment will be added or replaced for a data source. The *server-name* must identify a data source that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.SERVERS catalog view for the row that describes the data source.

SERVER OPTION *server-option-name* **FOR** *remote-server*
Indicates a comment will be added or replaced for a server option.

server-option-name
Identifies a server option. This option must be one that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.SERVEROPTIONS catalog view for the row that describes the server option.

remote-server
Describes the data source to which the *server-option* applies.

SERVER *server-name*
Names the data source to which the *server-option* applies. The *server-name* must identify a data source that is described in the catalog.

TYPE *server-type*
Specifies the type of data source—for example, DB2 Universal Database for OS/390 or Oracle—to which the *server-option* applies. The *server-type* can be specified in either lower- or uppercase; it will be stored in uppercase in the catalog.

VERSION
Specifies the version of the data source identified by *server-name*.

version
Specifies the version number. *version* must be an integer.

release
Specifies the number of the release of the version denoted by *version. release* must be an integer.

mod
Specifies the number of the modification of the release denoted by *release. mod* must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i');

or it can be the concatenated values of *version*, *release*, and, if applicable, *mod* (for example, '8.0.3').

WRAPPER *wrapper-name*
Identifies the wrapper that is used to access the data source referenced by *server-name*.

TABLE *table-name* or *view-name*
Indicates a comment will be added or replaced for a table or view. The *table-name* or *view-name* must identify a table or view (not an alias or nickname) that is described in the catalog (SQLSTATE 42704) and must not identify a declared temporary table (SQLSTATE 42995). The comment replaces the value for the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the table or view.

TABLESPACE *tablespace-name*
Indicates a comment will be added or replaced for a table space. The *tablespace-name* must identify a distinct table space that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TABLESPACES catalog view for the row that describes the tablespace.

TRIGGER *trigger-name*
Indicates a comment will be added or replaced for a trigger. The *trigger-name* must identify a distinct trigger that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TRIGGERS catalog view for the row that describes the trigger.

TYPE *type-name*
Indicates a comment will be added or replaced for a user-defined type. The *type-name* must identify a user-defined type that is described in the catalog (SQLSTATE 42704). If DISTINCT is specified, *type-name* must identify a distinct type that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.DATATYPES catalog view for the row that describes the user-defined type.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

TYPE MAPPING *type-mapping-name*
Indicates a comment will be added or replaced for a user-defined data type mapping. The *type-mapping-name* must identify a data type mapping that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TYPEMAPPINGS catalog view for the row that describes the mapping.

WRAPPER *wrapper-name*
Indicates a comment will be added or replaced for a wrapper. The *wrapper-name* must identify a wrapper that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.WRAPPERS catalog view for the row that describes the wrapper.

IS *string-constant*
Specifies the comment to be added or replaced. The *string-constant* can be any character string constant of up to 254 bytes. (Carriage return and line feed each count as 1 byte.)

table-name | *view-name* **(** { *column-name* **IS** *string-constant* } **...)**
This form of the COMMENT statement provides the ability to specify comments for multiple columns of a table or view. The column names must not be qualified, each name must identify a column of the specified table or view, and the table or view must be described in the catalog. The *table-name* cannot be a declared temporary table (SQLSTATE 42995).

A comment cannot be made on a column of an inoperative view (SQLSTATE 51024).

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NODEGROUP can be specified in place of DATABASE PARTITION GROUP

Examples:

Example 1: Add a comment for the EMPLOYEE table.

```
COMMENT ON TABLE EMPLOYEE
  IS 'Reflects first quarter 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 EMPLOYEE table.

```
COMMENT ON EMPLOYEE
  (WORKDEPT IS 'see DEPARTMENT table for names',
  EDLEVEL IS 'highest grade level passed in school' )
```

Example 5: Pellow wants to comment on the CENTRE function, which he created in his PELLOW schema, using the signature to identify the specific function to be commented on.

```
COMMENT ON FUNCTION CENTRE (INT,FLOAT)
  IS 'Frank''s CENTRE fctn, uses Chebychev method'
```

Example 6: McBride wants to comment on another CENTRE function, which she created in the PELLOW schema, using the specific name to identify the function instance to be commented on:

```
COMMENT ON SPECIFIC FUNCTION PELLOW.FOCUS92 IS
  'Louise''s most triumphant CENTRE function, uses the
       Brownian fuzzy-focus technique'
```

Example 7: Comment on the function ATOMIC_WEIGHT in the CHEM schema, where it is known that there is only one function with that name:

```
COMMENT ON FUNCTION CHEM.ATOMIC_WEIGHT
  IS 'takes atomic nbr, gives atomic weight'
```

Example 8: Eigler wants to comment on the SEARCH procedure, which he created in his EIGLER schema, using the signature to identify the specific procedure to be commented on.

```
COMMENT ON PROCEDURE SEARCH (CHAR,INT)
  IS 'Frank''s mass search and replace algorithm'
```

Example 9: Macdonald wants to comment on another SEARCH function, which he created in the EIGLER schema, using the specific name to identify the procedure instance to be commented on:

```
COMMENT ON SPECIFIC PROCEDURE EIGLER.DESTROY IS
  'Patrick''s mass search and destroy algorithm'
```

Example 10: Comment on the procedure OSMOSIS in the BIOLOGY schema, where it is known that there is only one procedure with that name:

```
COMMENT ON PROCEDURE BIOLOGY.OSMOSIS
  IS 'Calculations modelling osmosis'
```

Example 11: Comment on an index specification named INDEXSPEC.

```
COMMENT ON INDEX INDEXSPEC
  IS 'An index specification that indicates to the optimizer
  that the table referenced by nickname NICK1 has an index.'
```

Example 12: Comment on the wrapper whose default name is NET8.

```
COMMENT ON WRAPPER NET8
  IS 'The wrapper for data sources associated with
      Oracle's Net8 client software.'
```

COMMIT

The COMMIT statement terminates 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 through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

None required.

Syntax:

Description:
The unit of work in which the COMMIT statement is executed is terminated and a new unit of work is initiated. All changes made by the following statements executed during the unit of work are committed: ALTER, COMMENT ON, CREATE, DELETE, DROP, GRANT, INSERT, LOCK TABLE, REVOKE, SET INTEGRITY, SET transition-variable, and UPDATE.

The following statements, however, are not under transaction control and changes made by them are independent of the COMMIT statement:

- SET CONNECTION
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT PACKAGESET
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET EVENT MONITOR STATE
- SET PASSTHRU

Note: Although the SET PASSTHRU statement is not under transaction control, the passthru session initiated by the statement is under transaction control.

- SET PATH
- SET SCHEMA
- SET SERVER OPTION

All locks acquired by the unit of work subsequent to its initiation are released, except necessary locks for open cursors that are declared WITH HOLD.All open cursors not defined WITH HOLD are closed. Open cursors defined WITH HOLD remain open, and the cursor is positioned before the next logical row of the result table. (A FETCH must be performed before a positioned UPDATE or DELETE statement is issued.) 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 savepoints set within the transaction are released.

Notes:

- It is strongly recommended that each application process explicitly ends its unit of work before terminating. If the application program ends normally without a COMMIT or ROLLBACK statement then the database manager attempts a commit or rollback depending on the application environment.
- For information on the impact of COMMIT on cached dynamic SQL statements, see "EXECUTE".
- For information on potential impacts of COMMIT on declared temporary tables, see "DECLARE GLOBAL TEMPORARY TABLE".

Example:

Commit alterations to the database made since the last commit point.

```
COMMIT WORK
```

Related reference:

- "EXECUTE" on page 544
- "DECLARE GLOBAL TEMPORARY TABLE" on page 488

Related samples:

- "dynamic.sqb -- How to update table data with cursor dynamically (MF COBOL)"
- "tbconstr.sqc -- How to create, use, and drop constraints (C)"
- "tbsavept.sqc -- How to use external savepoints (C)"
- "tut_mod.sqc -- How to modify table data (C)"
- "tut_use.sqc -- How to modify a database (C)"
- "tbconstr.sqC -- How to create, use, and drop constraints (C++)"

- “tut_mod.sqC -- How to modify table data (C++)”
- “tut_use.sqC -- How to modify a database (C++)”

Compound SQL (Dynamic)

A compound statement groups other statements together into an executable block. SQL variables can be declared within a dynamically prepared atomic compound statement.

Invocation:

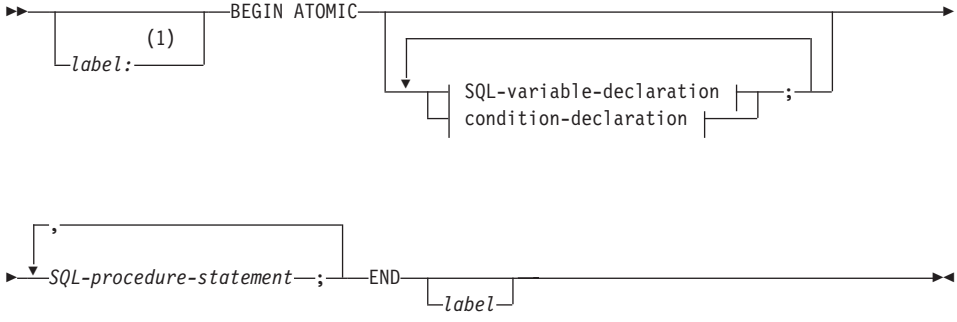
This statement can be embedded in a trigger, SQL function, or SQL method, or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

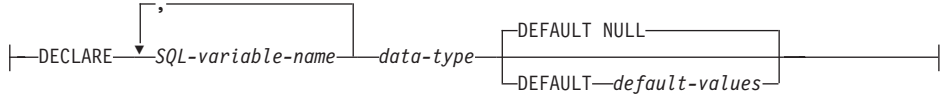
No privileges are required to invoke a dynamic compound statement. However, the authorization ID of the compound statement must hold the necessary privileges to invoke the SQL statements embedded in the compound statement.

Syntax:

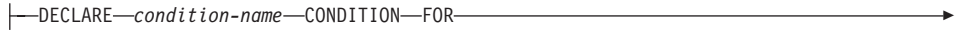
dynamic-compound-statement

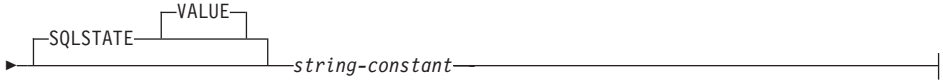


SQL-variable-declaration:



condition-declaration:





Notes:

1 A label can only be specified when the statement is in a function, method, or trigger definition.

Description:

label
Defines the label for the code block. If the beginning label is specified, it can be used to qualify SQL variables declared in the dynamic compound statement and can also be specified on a LEAVE statement. If the ending label is specified, it must be the same as the beginning label.

ATOMIC
ATOMIC indicates that, if an error occurs in the compound statement, all SQL statements in the compound statement will be rolled back and any remaining SQL statements in the compound statement are not processed.

SQL-procedure-statement
The following list of *SQL-control-statements* can be used within the dynamic compound statement:

- FOR Statement
- GET DIAGNOSTICS Statement
- IF Statement
- ITERATE Statement
- LEAVE Statement
- SIGNAL Statement
- WHILE Statement

The SQL statements that can be issued are:

- fullselect (A common-table-expression can precede the fullselect.)
- Searched UPDATE
- Searched DELETE
- INSERT
- SET variable statement

(Searched UPDATE, searched DELETE, and INSERT on nicknames inside compound SQL is not supported.)

SQL-variable-declaration
Declares a variable that is local to the dynamic compound statement.

SQL-variable-name
Defines the name of a local variable. DB2 converts all SQL variable names to uppercase. The name cannot:

- be the same as another SQL variable within the compound statement
- be the same as a parameter name

If an SQL statement contains an identifier with the same name as an SQL variable and a column reference, DB2 interprets the identifier as a column.

data-type
Specifies the data type of the variable.

DEFAULT *default-values* **or NULL**
Defines the default for the SQL variable. The variable is initialized when the dynamic compound statement is called. If a default value is not specified, the variable is initialized to NULL.

condition-declaration
Declares a condition name and corresponding SQLSTATE value.

condition-name
Specifies the name of the condition. The condition name must be unique within the procedure body and can be referenced only within the compound statement in which it is declared.

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE associated with the condition. The *string-constant* must be specified as five characters enclosed in single quotes, and cannot be '00000'.

Notes:

- Dynamic compound statements are compiled by DB2 as one single statement. This statement is effective for short scripts involving little control flow logic but significant dataflow. For larger constructs with requirements for nested complex control flow or condition handling, a better choice is to use SQL procedures. For more details on using SQL procedures, see "CREATE PROCEDURE".

Examples:

Example 1:

This example illustrates how inline SQL PL can be used in a data warehousing scenario for data cleansing.

The example introduces three tables. The ″target″ table contains the cleansed data. The ″except″ table stores rows that cannot be cleansed (exceptions) and the ″source″ table contains the raw data to be cleansed.

A simple SQL function called ″discretize″ is used to classify and modify the data. It returns NULL for all bad data. The Dynamic Compound statement then cleanses the data. It walks all rows of the source table in a FOR-loop and decides whether the current row gets inserted into the ″target″ or the ″except″ table, depending on the result of the ″discretize″ function. More elaborate mechanisms (multistage cleansing) are possible with this technique.

The same code can be written using an SQL Procedure or any other procedure or application in a host language. However, the dynamic compound statement offers a unique advantage in that the FOR-loop does not open a cursor and the single row inserts are not really single row inserts. In fact, the logic is effectively a multi-table insert from a shared select.

This is achieved by compilation of the dynamic compound as a single statement. Similar to a view whose body is integrated into the query that uses it and then is compiled and optimized as a whole within the query context, the DB2 optimizer compiles and optimizes both the control and data flow together. The whole logic is therefore executed within DB2's runtime. No data is moved outside of the core DB2 engine, as would be done for a stored procedure.

The first step is to create the required tables:

```
CREATE TABLE target
  (pk INTEGER NOT NULL
  PRIMARY KEY, c1 INTEGER)
```

This creates a table called TARGET to contain the cleansed data.

```
CREATE TABLE except
  (pk INTEGER NOT NULL
  PRIMARY KEY, c1 INTEGER)
```

This creates a table called EXCEPT to contain the exceptions.

```
CREATE TABLE source
  (pk INTEGER NOT NULL
  PRIMARY KEY, c1 INTEGER)
```

This creates a table called SOURCE to hold the data that is to be cleansed.

Next, we create a ″discretize″ function to cleanse the data by throwing out all values outside [0..1000] and aligning them to steps of 10.

```
CREATE FUNCTION discretize(raw INTEGER) RETURNS INTEGER
  RETURN CASE
    WHEN raw < 0 THEN CAST(NULL AS INTEGER)
    WHEN raw > 1000 THEN NULL
    ELSE ((raw / 10) * 10) + 5
  END
```

Then we insert the values:

```
INSERT INTO source (pk, c1)
  VALUES (1,   -5),
         (2, NULL),
         (3, 1200),
         (4,   23),
         (5,   10),
         (6,  876)
```

Invoke the function:

```
BEGIN ATOMIC
  FOR row AS
    SELECT pk, c1, discretize(c1) AS d FROM source
  DO
    IF row.d is NULL THEN
      INSERT INTO except VALUES(row.pk, row.c1);
    ELSE
      INSERT INTO target VALUES(row.pk, row.d);
    END IF;
  END FOR;
END
```

And test the results:

```
SELECT * FROM except ORDER BY 1
PK          C1
----------- -----------
          1          -5
          2           -
          3        1200
  3 record(s) selected.

SELECT * FROM target ORDER BY 1
PK          C1
----------- -----------
          4          25
          5          15
          6         875
  3 record(s) selected.
```

The final step is to clean up:

```
DROP FUNCTION discretize
DROP TABLE source
DROP TABLE target
DROP TABLE except
```

Related reference:

- "CREATE PROCEDURE" on page 296

Related samples:

- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"

Compound SQL (Embedded)

Combines one or more other SQL statements (*sub-statements*) into an executable block.

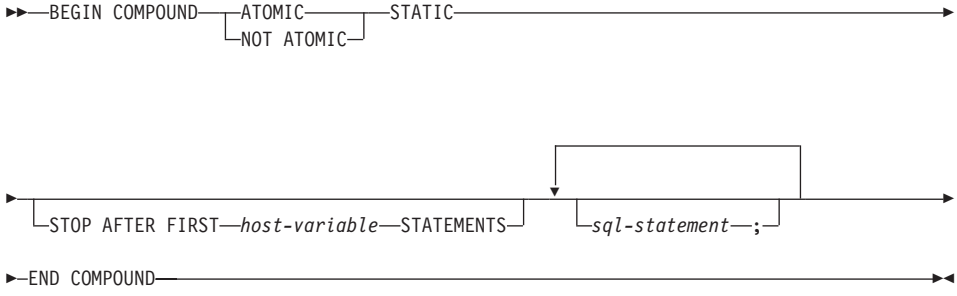
Invocation:

This statement can only be embedded in an application program. The entire Compound SQL statement construct is an executable statement that cannot be dynamically prepared. The statement is not supported in REXX.

Authorization:

None for the Compound SQL statement itself. The authorization ID of the Compound SQL statement must have the appropriate authorization on all the individual statements that are contained within the Compound SQL statement.

Syntax:



Description:

ATOMIC
Specifies that, if any of the sub-statements within the Compound SQL statement fail, then all changes made to the database by any of the sub-statements, including changes made by successful sub-statements, are undone.

NOT ATOMIC
Specifies that, regardless of the failure of any sub-statements, the Compound SQL statement will not undo any changes made to the database by the other sub-statements.

STATIC
Specifies that input variables for all sub-statements retain their original value. For example, if

```
SELECT ... INTO :abc ...
```

is followed by:

```
UPDATE T1 SET C1 = 5 WHERE C2 = :abc
```

the UPDATE statement will use the value that :abc had at the start of the execution of the Compound SQL statement, not the value that follows the SELECT INTO.

If the same variable is set by more than one sub-statement, the value of that variable following the Compound SQL statement is the value set by the last sub-statement.

Note: Non-static behavior is not supported. This means that the sub-statements should be viewed as executing non-sequentially and sub-statements should not have interdependencies.

STOP AFTER FIRST
Specifies that only a certain number of sub-statements will be executed.

host-variable
A small integer that specifies the number of sub-statements to be executed.

STATEMENTS
Completes the STOP AFTER FIRST *host-variable* clause.

sql-statement
All executable statements except the following can be contained within an embedded static compound SQL statement:

CALL
CLOSE
CONNECT
Compound SQL
DESCRIBE
DISCONNECT
EXECUTE IMMEDIATE
FETCH
OPEN
PREPARE
RELEASE (Connection)
RELEASE SAVEPOINT
ROLLBACK
SAVEPOINT
SET CONNECTION

Note: INSERT, UPDATE, and DELETE are not supported in compound SQL for use with nicknames.

If a COMMIT statement is included, it must be the last sub-statement. If COMMIT is in this position, it will be issued even if the STOP AFTER FIRST *host-variable* STATEMENTS clause indicates that not all of the sub-statements are to executed. For example, suppose COMMIT is the last

sub-statement in a compound SQL block of 100 sub-statements. If the STOP AFTER FIRST STATEMENTS clause indicates that only 50 sub-statements are to be executed, then COMMIT will be the 51st sub-statement.

An error will be returned if COMMIT is included when using CONNECT TYPE 2 or running in an XA distributed transaction processing environment (SQLSTATE 25000).

Rules:

- DB2 Connect does not support SELECT statements selecting LOB columns in a compound SQL block.
- No host language code is allowed within a Compound SQL statement; that is, no host language code is allowed between the sub-statements that make up the Compound SQL statement.
- Only NOT ATOMIC Compound SQL statements will be accepted by DB2 Connect.
- Compound SQL statements cannot be nested.
- An Atomic Compound SQL statement cannot be issued inside a savepoint (SQLSTATE 3B002).
- A prepared COMMIT statement is not allowed in an ATOMIC Compound SQL statement

Notes:

One SQLCA is returned for the entire Compound SQL statement. Most of the information in that SQLCA reflects the values set by the application server when it processed the last sub-statement. For instance:

- The SQLCODE and SQLSTATE are normally those for the last sub-statement (the exception is described in the next point).
- If a 'no data found' warning (SQLSTATE '02000') is returned, that warning is given precedence over any other warning so that a WHENEVER NOT FOUND exception can be acted upon. (This means that the SQLCODE, SQLERRML, SQLERRMC, and SQLERRP fields in the SQLCA that is eventually returned to the application are those from the sub-statement that triggered the 'no data found' warning. If there is more than one 'no data found' warning within the Compound SQL statement, the fields for the last sub-statement will be the fields that are returned.)
- The SQLWARN indicators are an accumulation of the indicators set for all sub-statements.

If one or more errors occurred during NOT ATOMIC Compound SQL execution and none of these are of a serious nature, the SQLERRMC will contain information on up to a maximum of seven of these errors. The first

token of the SQLERRMC will indicate the total number of errors that occurred. The remaining tokens will each contain the ordinal position and the SQLSTATE of the failing sub-statement within the Compound SQL statement. The format is a character string of the form:

```
nnnXssscccccc
```

with the substring starting with X repeating up to six more times and the string elements defined as follows.

nnn The total number of statements that produced errors. (If the number would exceed 999, counting restarts at zero.) This field is left-justified and padded with blanks.

X The token separator X'FF'.

sss The ordinal position of the statement that caused the error. (If the number would exceed 999, counting restarts at zero.) For example, if the first statement failed, this field would contain the number one left-justified ('1 ').

ccccc The SQLSTATE of the error.

The second SQLERRD field contains the number of statements that failed (returned negative SQLCODEs).

The third SQLERRD field in the SQLCA is an accumulation of the number of rows affected by all sub-statements.

The fourth SQLERRD field in the SQLCA is a count of the number of successful sub-statements. If, for example, the third sub-statement in a Compound SQL statement failed, the fourth SQLERRD field would be set to 2, indicating that 2 sub-statements were successfully processed before the error was encountered.

The fifth SQLERRD field in the SQLCA is an accumulation of the number of rows updated or deleted due to the enforcement of referential integrity constraints for all sub-statements that triggered such constraint activity.

Examples:

Example 1: In a C program, issue a Compound SQL statement that updates both the ACCOUNTS and TELLERS tables. If there is an error in any of the statements, undo the effect of all statements (ATOMIC). If there are no errors, commit the current unit of work.

```
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
  UPDATE ACCOUNTS SET ABALANCE = ABALANCE + :delta
    WHERE AID = :aid;
  UPDATE TELLERS SET TBALANCE = TBALANCE + :delta
```

```
    WHERE TID = :tid;
  INSERT INTO TELLERS (TID, BID, TBALANCE) VALUES (:i, :branch_id, 0);
  COMMIT;
END COMPOUND;
```

Example 2: In a C program, insert 10 rows of data into the database. Assume the host variable :nbr contains the value 10 and S1 is a prepared INSERT statement. Further, assume that all the inserts should be attempted regardless of errors (NOT ATOMIC).

```
EXEC SQL BEGIN COMPOUND NOT ATOMIC STATIC STOP AFTER FIRST :nbr STATEMENTS
  EXECUTE S1 USING DESCRIPTOR :*sqlda0;
  EXECUTE S1 USING DESCRIPTOR :*sqlda1;
  EXECUTE S1 USING DESCRIPTOR :*sqlda2;
  EXECUTE S1 USING DESCRIPTOR :*sqlda3;
  EXECUTE S1 USING DESCRIPTOR :*sqlda4;
  EXECUTE S1 USING DESCRIPTOR :*sqlda5;
  EXECUTE S1 USING DESCRIPTOR :*sqlda6;
  EXECUTE S1 USING DESCRIPTOR :*sqlda7;
  EXECUTE S1 USING DESCRIPTOR :*sqlda8;
  EXECUTE S1 USING DESCRIPTOR :*sqlda9;
END COMPOUND;
```

Related reference:

- "Compound statement (Procedure)" on page 767

Related samples:

- "dbuse.sqc -- How to use a database (C)"
- "dbuse.sqC -- How to use a database (C++)"

CONNECT (Type 1)

The CONNECT (Type 1) statement connects an application process to the identified application server according to the rules for remote unit of work.

An application process can only be connected to one application server at a time. This is called the *current server*. A default application server may be established when the application requester is initialized. If implicit connect is available and an application process is started, it is implicitly connected to the default application server. The application process can explicitly connect to a different application server by issuing a CONNECT TO statement. A connection lasts until a CONNECT RESET statement or a DISCONNECT statement is issued or until another CONNECT TO statement changes the application server.

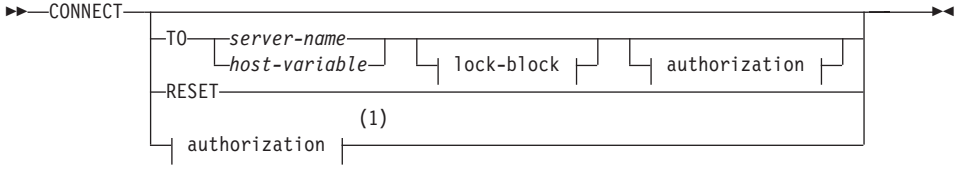
Invocation:

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.

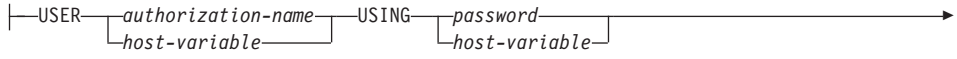
Authorization:

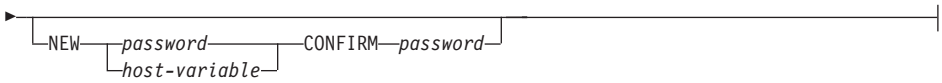
The authorization ID of the statement must be authorized to connect to the identified application server. Depending on the authentication setting for the database, the authorization check may be performed by either the client or the server. For a partitioned database, the user and group definitions must be identical across partitions.

Syntax:

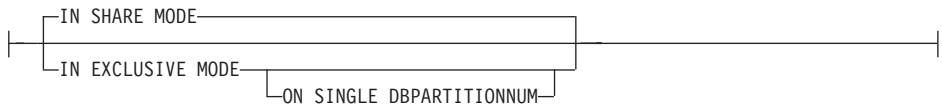


authorization:





lock-block:



Notes:

1 This form is only valid if implicit connect is enabled.

Description:

CONNECT (with no operand)
Returns information about the current server. The information is returned in the SQLERRP field of the SQLCA as described in "Successful Connection".

If a connection state exists, the authorization ID and database alias are placed in the SQLERRMC field of the SQLCA. If the authorization ID is longer than 8 bytes, it will be truncated to 8 bytes, and the truncation will be flagged in the SQLWARN0 and SQLWARN1 fields of the SQLCA, with 'W' and 'A', respectively. If the database configuration parameter DYN_QUERY_MGMT is enabled, then the SQLWARN0 and SQLWARN7 fields of the SQLCA will be flagged with 'W' and 'E', respectively.

If no connection exists and implicit connect is possible, then an attempt to make an implicit connection is made. If implicit connect is not available, this attempt results in an error (no existing connection). If no connection, then the SQLERRMC field is blank.

The territory code and code page of the application server are placed in the SQLERRMC field (as they are with a successful CONNECT TO statement).

This form of CONNECT:

- Does not require the application process to be in the connectable state.
- If connected, does not change the connection state.
- If unconnected and implicit connect is available, a connection to the default application server is made. In this case, the country/region code and code page of the application server are placed in the SQLERRMC field, like a successful CONNECT TO statement.
- If unconnected and implicit connect is not available, the application process remains unconnected.
- Does not close cursors.

TO *server-name* or *host-variable*
Identifies the application server by the specified *server-name* or a *host-variable* which contains the server-name.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-justified and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

Note: DB2 UDB for OS/390 and z/OS supports a 16-byte location name, and DB2 UDB for iSeries supports an 18-byte target database name. DB2 Version 8 only supports the use of an 8-byte database alias name on the SQL CONNECT statement. However, the database alias name can be mapped to an 18-byte database name through the Database Connection Service Directory.

When the CONNECT TO statement is executed, the application process must be in the connectable state.

Successful Connection:

If the CONNECT TO statement is successful:

- All open cursors are closed, all prepared statements are destroyed, 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 actual name of the application server (not an alias) is placed in the CURRENT SERVER special register.
- Information about the application server is placed in the SQLERRP field of the SQLCA. If the application server is an IBM product, the information has the form *pppvvrrm*, where:
  - *ppp* identifies the product as follows:
    - DSN for DB2 UDB for OS/390 and z/OS
    - ARI for DB2 Server for VSE & VM
    - QSQ for DB2 UDB for iSeries
    - SQL for DB2 UDB for UNIX and Windows
  - *vv* is a two-digit version identifier, such as '08'
  - *rr* is a two-digit release identifier, such as '01'
  - *m* is a one-digit modification level identifier, such as '0'.

This release (Version 8) of DB2 UDB for UNIX and Windows is identified as 'SQL08010'.

- The SQLERRMC field of the SQLCA is set to contain the following values (separated by X'FF')
  1. the country/region code of the application server (or blanks if using DB2 Connect),
  2. the code page of the application server (or CCSID if using DB2 Connect),
  3. the authorization ID (up to first 8 bytes only),
  4. the database alias,
  5. the platform type of the application server. Currently identified values are:

Token	Server
QAS	DB2 Universal Database for iSeries
QDB2	DB2 Universal Database for OS/390 and z/OS
QDB2/2	DB2 Universal Database for OS/2
QDB2/6000	DB2 Universal Database for AIX
QDB2/HPUX	DB2 Universal Database for HP-UX
QDB2/LINUX	DB2 Universal Database for Linux
QDB2/NT	DB2 Universal Database for Windows NT, 2000, and XP
QDB2/SUN	DB2 Universal Database for Solaris Operating System
QSQLDS/VM	DB2 Server for VM
QSQLDS/VSE	DB2 Server for VSE

  6. The agent ID. It identifies the agent executing within the database manager on behalf of the application. This field is the same as the agent_id element returned by the database monitor.
  7. The agent index. It identifies the index of the agent and is used for service.
  8. Partition number. For a non-partitioned database, this is always 0, if present.
  9. The code page of the application client.
  10. Number of partitions in a partitioned database. If the database cannot be partitioned, the value is 0 (zero). Token is present only with Version 5 or later.

- The SQLERRD(1) field of the SQLCA indicates the maximum expected difference in length of mixed character data (CHAR data types) when converted to the database code page from the application code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction.
- The SQLERRD(2) field of the SQLCA indicates the maximum expected difference in length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction.
- The SQLERRD(3) field of the SQLCA indicates whether or not the database on the connection is updatable. A database is initially updatable, but is changed to read-only if a unit of work determines the authorization ID cannot perform updates. The value is one of:
  - 1 - updatable
  - 2 - read-only
- The SQLERRD(4) field of the SQLCA returns certain characteristics of the connection. The value is one of:

  0 - N/A (only possible if running from a down-level client which is one phase commit and is an updater).

  1 - one-phase commit.

  2 - one-phase commit; read-only (only applicable to connections to DRDA1 databases in TP Monitor environment).

  3 - two-phase commit.
- The SQLERRD(5) field of the SQLCA returns the authentication type of the connection. The value is one of:

  0 - Authenticated on the server.

  1 - Authenticated on the client.

  2 - Authenticated using DB2 Connect.

  3 - Authenticated using Distributed Computing Environment security services.

  255 - Authentication not specified.
- The SQLERRD(6) field of the SQLCA returns the partition number of the partition to which the connection was made if the database is partitioned. Otherwise, a value of 0 is returned.
- The SQLWARN1 field in the SQLCA will be set to 'A' if the authorization ID of the successful connection is longer than 8 bytes.

This indicates that truncation has occurred. The SQLWARN0 field in the SQLCA will be set to 'W' to indicate this warning.

- The SQLWARN7 field in the SQLCA will be set to 'E' if the database configuration parameter DYN_QUERY_MGMT for the database is enabled. The SQLWARN0 field in the SQLCA will be set to 'W' to indicate this warning.

Unsuccessful Connection:

If the CONNECT TO 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. The first three characters of the module name identify the product.
- If the CONNECT TO 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 TO statement is unsuccessful because the *server-name* is not listed in the local directory, an error message (SQLSTATE 08001) is issued and the connection state of the application process remains unchanged:
  - If the application requester was not connected to an application server then the application process remains unconnected.
  - If the application requester was already connected to an application server, the application process remains connected to that application server. Any further statements are executed at that application server.
- If the CONNECT TO statement is unsuccessful for any other reason, the application process is placed into the unconnected state.

IN SHARE MODE
Allows other concurrent connections to the database and prevents other users from connecting to the database in exclusive mode.

IN EXCLUSIVE MODE
Prevents concurrent application processes from executing any operations at the application server, unless they have the same authorization ID as the user holding the exclusive lock. This option is not supported by DB2 Connect.

ON SINGLE DBPARTITIONNUM
Specifies that the coordinator database partition is connected in exclusive mode and all other database partitions are connected in share mode. This option is only effective in a partitioned database.

RESET
Disconnects the application process from the current server. A commit

operation is performed. If implicit connect is available, the application process remains unconnected until an SQL statement is issued.

USER *authorization-name/host-variable*
Identifies the user ID trying to connect to the application server. If a host-variable is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The user ID that is contained within the *host-variable* must be left justified and must not be delimited by quotation marks.

USING *password/host-variable*
Identifies the password of the user ID trying to connect to the application server. *Password* or *host-variable* may be up to 18 characters. If a host variable is specified, it must be a character string variable with a length attribute not greater than 18 and it must not include an indicator variable.

NEW *password/host-variable* **CONFIRM** *password*
Identifies the new password that should be assigned to the user ID identified by the USER option. *Password* or *host-variable* may be up to 18 characters. If a host variable is specified, it must be a character string variable with a length attribute not greater than 18 and it must not include an indicator variable. The system on which the password will be changed depends on how user authentication is set up.

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NODE can be specified in place of DBPARTITIONNUM
- It is good practice for the first SQL statement executed by an application process to be the CONNECT TO statement.
- If a CONNECT TO statement is issued to the current application server with a different user ID and password then the conversation is deallocated and reallocated. All cursors are closed by the database manager (with the loss of the cursor position if the WITH HOLD option was used).
- If a CONNECT TO statement is issued to the current application server with the same user ID and password then the conversation is not deallocated and reallocated. Cursors, in this case, are not closed.
- To use a multiple-partition database environment, the user or application must connect to one of the partitions listed in the `db2nodes.cfg` file. You should try to ensure that not all users use the same partition as the coordinator partition.

Examples:

Example 1: In a C program, connect to the application server TOROLAB, using database alias TOROLAB, user ID FERMAT, and password THEOREM.

```
EXEC SQL  CONNECT TO TOROLAB USER FERMAT USING THEOREM;
```

Example 2: In a C program, connect to an application server whose database alias is stored in the host variable APP_SERVER (varchar(8)). Following a successful connection, copy the 3-character product identifier of the application server to the variable PRODUCT (char(3)).

```
EXEC SQL  CONNECT TO :APP_SERVER;
if (strncmp(SQLSTATE,'00000',5))
  strncpy(PRODUCT,sqlca.sqlerrp,3);
```

Related concepts:

- "Distributed relational databases" in the *SQL Reference, Volume 1*
- "Data partitioning across multiple partitions" in the *SQL Reference, Volume 1*

Related samples:

- "advsql.sqb -- How to read table data using CASE (MF COBOL)"
- "dbmcon.sqc -- How to use multiple databases (C)"
- "dbmcon.sqC -- How to use multiple databases (C++)"

CONNECT (Type 2)

The CONNECT (Type 2) statement connects an 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.

Most aspects of a CONNECT (Type 1) statement also apply to a CONNECT (Type 2) statement. Rather than repeating that material here, this section describes only those elements of Type 2 that differ from Type 1.

Invocation:

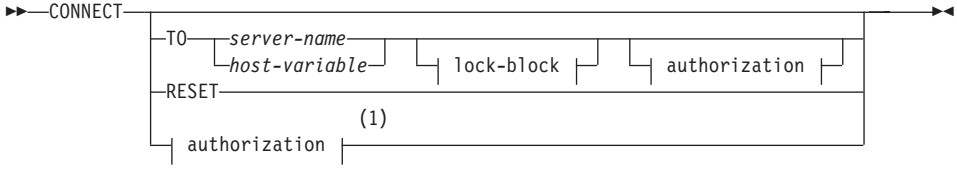
Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

Authorization:

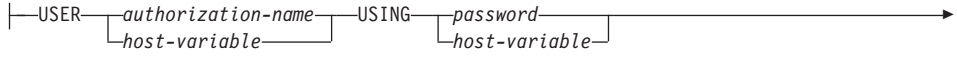
The authorization ID of the statement must be authorized to connect to the identified application server. Depending on the authentication setting for the database, the authorization check may be performed by either the client or the server. For a partitioned database, the user and group definitions must be identical across partitions.

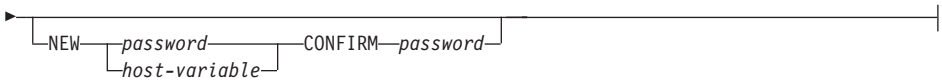
Syntax:

The selection between Type 1 and Type 2 is determined by precompiler options. For an overview of these options, see "Distributed relational databases".

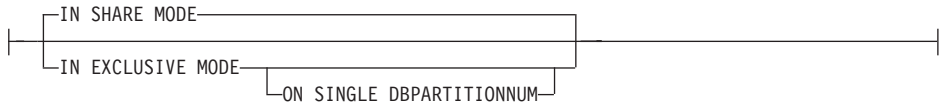


authorization:





lock-block:



Notes:

1 This form is only valid if implicit connect is enabled.

Description:

TO *server-name/host-variable*
The rules for coding the name of the server are the same as for Type 1.

If the SQLRULES(STD) option is in effect, the *server-name* must not identify an existing connection of the application process, otherwise an error (SQLSTATE 08002) is raised.

If the SQLRULES(DB2) option is in effect and the *server-name* identifies an existing connection of the application process, that connection is made current and the old connection is placed into the dormant state. That is, the effect of the CONNECT statement in this situation is the same as that of a SET CONNECTION statement.

For information about the specification of SQLRULES, see "Options that Govern Distributed Unit of Work Semantics".

Successful Connection

If the CONNECT TO statement is successful:

- A connection to the application server is either created (or made non-dormant) and placed into the current and held states.
- If the CONNECT TO is directed to a different server than the current server, then the current connection is placed into the dormant state.
- The CURRENT SERVER special register and the SQLCA are updated in the same way as for CONNECT (Type 1).

Unsuccessful Connection

If the CONNECT TO statement is unsuccessful:

- No matter what the reason for failure, the connection state of the application process and the states of its connections are unchanged.

- As with an unsuccessful Type 1 CONNECT, the SQLERRP field of the SQLCA is set to the name of the module at the application requester or server that detected the error.

CONNECT (with no operand)**, IN SHARE/EXCLUSIVE MODE, USER,** and **USING**

If a connection exists, Type 2 behaves like a Type 1. The authorization ID and database alias are placed in the SQLERRMC field of the SQLCA. If a connection does not exist, no attempt to make an implicit connection is made and the SQLERRP and SQLERRMC fields return a blank. (Applications can check if a current connection exists by checking these fields.)

A CONNECT with no operand that includes USER and USING can still connect an application process to a database using the DB2DBDFT environment variable. This method is equivalent to a Type 2 CONNECT RESET, but permits the use of a user ID and password.

RESET

Equivalent to an explicit connect to the default database if it is available. If a default database is not available, the connection state of the application process and the states of its connections are unchanged.

Availability of a default database is determined by installation options, environment variables, and authentication settings.

Rules:

- As outlined in "Options that Govern Distributed Unit of Work Semantics", a set of connection options governs the semantics of connection management. Default values are assigned to every preprocessed source file. An application can consist of multiple source files precompiled with different connection options.

  Unless a SET CLIENT command or API has been executed first, the connection options used when preprocessing the source file containing the first SQL statement executed at run time become the effective connection options.

  If a CONNECT statement from a source file preprocessed with different connection options is subsequently executed without the execution of any intervening SET CLIENT command or API, an error (SQLSTATE 08001) is returned. Note that once a SET CLIENT command or API has been executed, the connection options used when preprocessing all source files in the application are ignored.

  Example 1 in the "Examples" section of this statement illustrates these rules.
- Although the CONNECT TO statement can be used to establish or switch connections, CONNECT TO with the USER/USING clause will only be accepted when there is no current or dormant connection to the named

server. The connection must be released before issuing a connection to the same server with the USER/USING clause, otherwise it will be rejected (SQLSTATE 51022). Release the connection by issuing a DISCONNECT statement or a RELEASE statement followed by a COMMIT statement.

Notes:

- Implicit connect is supported for the first SQL statement in an application with Type 2 connections. In order to execute SQL statements on the default database, first the CONNECT RESET or the CONNECT USER/USING statement must be used to establish the connection. The CONNECT statement with no operands will display information about the current connection if there is one, but will not connect to the default database if there is no current connection.

Comparing Type 1 and Type 2 CONNECT Statements:

The semantics of the CONNECT statement are determined by the CONNECT precompiler option or the SET CLIENT API (see "Options that Govern Distributed Unit of Work Semantics"). CONNECT Type 1 or CONNECT Type 2 can be specified and the CONNECT statements in those programs are known as Type 1 and Type 2 CONNECT statements, respectively. Their semantics are described below:

Use of **CONNECT TO**:

Type 1	Type 2
Each unit of work can only establish connection to one application server.	Each unit of work can establish connection to multiple application servers.
The current unit of work must be committed or rolled back before allowing a connection to another application server.	The current unit of work need not be committed or rolled back before connecting to another application server.
The CONNECT statement establishes the current connection. Subsequent SQL requests are forwarded to this connection until changed by another CONNECT.	Same as Type 1 CONNECT if establishing the first connection. If switching to a dormant connection and SQLRULES is set to STD, then the SET CONNECTION statement must be used instead.
Connecting to the current connection is valid and does not change the current connection.	Same as Type 1 CONNECT if the SQLRULES precompiler option is set to DB2. If SQLRULES is set to STD, then the SET CONNECTION statement must be used instead.

Type 1	Type 2
Connecting to another application server disconnects the current connection. The new connection becomes the current connection. Only one connection is maintained in a unit of work.	Connecting to another application server puts the current connection into the *dormant state*. The new connection becomes the current connection. Multiple connections can be maintained in a unit of work. If the CONNECT is for an application server on a dormant connection, it becomes the current connection. Connecting to a dormant connection using CONNECT is only allowed if SQLRULES(DB2) was specified. If SQLRULES(STD) was specified, then the SET CONNECTION statement must be used instead.
SET CONNECTION statement is supported for Type 1 connections, but the only valid target is the current connection.	SET CONNECTION statement is supported for Type 2 connections to change the state of a connection from dormant to current.

Use of **CONNECT...USER...USING**:

Type 1	Type 2
Connecting with the USER...USING clauses disconnects the current connection and establishes a new connection with the given authorization name and password.	Connecting with the USER/USING clause will only be accepted when there is no current or dormant connection to the same named server.

Use of **Implicit CONNECT, CONNECT RESET**, and **Disconnecting**:

Type 1	Type 2
CONNECT RESET can be used to disconnect the current connection.	CONNECT RESET is equivalent to connecting to the default application server explicitly if one has been defined in the system. Connections can be disconnected by the application at a successful COMMIT. Prior to the commit, use the RELEASE statement to mark a connection as release-pending. All such connections will be disconnected at the next COMMIT. An alternative is to use the precompiler options DISCONNECT(EXPLICIT), DISCONNECT(CONDITIONAL), DISCONNECT(AUTOMATIC), or the DISCONNECT statement instead of the RELEASE statement.
After using CONNECT RESET to disconnect the current connection, if the next SQL statement is not a CONNECT statement, then it will perform an implicit connect to the default application server if one has been defined in the system.	CONNECT RESET is equivalent to an explicit connect to the default application server if one has been defined in the system.
It is an error to issue consecutive CONNECT RESETs.	It is an error to issue consecutive CONNECT RESETs ONLY if SQLRULES(STD) was specified because this option disallows the use of CONNECT to existing connection.
CONNECT RESET also implicitly commits the current unit of work.	CONNECT RESET does not commit the current unit of work.
If an existing connection is disconnected by the system for whatever reasons, then subsequent non-CONNECT SQL statements to this database will receive an SQLSTATE of 08003.	If an existing connection is disconnected by the system, COMMIT, ROLLBACK, and SET CONNECTION statements are still permitted.
The unit of work will be implicitly committed when the application process terminates successfully.	Same as Type 1.
All connections (only one) are disconnected when the application process terminates.	All connections (current, dormant, and those marked for release pending) are disconnected when the application process terminates.

CONNECT Failures:

Type 1	Type 2
Regardless of whether there is a current connection when a CONNECT fails (with an error other than server-name not defined in the local directory), the application process is placed in the unconnected state. Subsequent non-CONNECT statements receive an SQLSTATE of 08003.	If there is a current connection when a CONNECT fails, the current connection is unaffected. If there was no current connection when the CONNECT fails, then the program is then in an unconnected state. Subsequent non-CONNECT statements receive an SQLSTATE of 08003.

Examples:

Example 1:

This example illustrates the use of multiple source programs (shown in the boxes), some preprocessed with different connection options (shown above the code), and one of which contains a SET CLIENT API call.

PGM1: CONNECT(2) SQLRULES(DB2) DISCONNECT(CONDITIONAL)

```
...
exec sql CONNECT TO OTTAWA;
exec sql SELECT col1 INTO :hv1
FROM tbl1;
...
```

PGM2: CONNECT(2) SQLRULES(STD) DISCONNECT(AUTOMATIC)

```
...
exec sql CONNECT TO QUEBEC;
exec sql SELECT col1 INTO :hv1
FROM tbl2;
...
```

PGM3: CONNECT(2) SQLRULES(STD) DISCONNECT(EXPLICIT)

```
...
SET CLIENT CONNECT 2  SQLRULES DB2  DISCONNECT EXPLICIT [1]
exec sql CONNECT TO LONDON;
exec sql SELECT col1 INTO :hv1
FROM tbl3;
...
```

1 Note: not the actual syntax of the SET CLIENT API

PGM4: CONNECT(2) SQLRULES(DB2) DISCONNECT(CONDITIONAL)

```
...
exec sql CONNECT TO REGINA;
exec sql SELECT col1 INTO :hv1
FROM tbl4;
...
```

If the application executes PGM1 then PGM2:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to QUEBEC fails with SQLSTATE 08001 because both SQLRULES and DISCONNECT are different.

If the application executes PGM1 then PGM3:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to LONDON runs: connect=2, sqlrules=DB2, disconnect=EXPLICIT

This is OK because the SET CLIENT API is run before the second CONNECT statement.

If the application executes PGM1 then PGM4:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to REGINA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL

This is OK because the preprocessor options for PGM1 are the same as those for PGM4.

Example 2:

This example shows the interrelationships of the CONNECT (Type 2), SET CONNECTION, RELEASE, and DISCONNECT statements. S0, S1, S2, and S3 represent four servers.

Sequence	Statement	Current Server	Dormant Connections	Release Pending
0	No statement	None	None	None
1	SELECT * FROM TBLA	S0 (default)	None	None
2	CONNECT TO S1	S1	S0	None
	SELECT * FROM TBLB	S1	S0	None
3	CONNECT TO S2	S2	S0, S1	None
	UPDATE TBLC SET ...	S2	S0, S1	None

Sequence	Statement	Current Server	Dormant Connections	Release Pending
4	CONNECT TO S3	S3	S0, S1, S2	None
	SELECT * FROM TBLD	S3	S0, S1, S2	None
5	SET CONNECTION S2	S2	S0, S1, S3	None
6	RELEASE S3	S2	S0, S1	S3
7	COMMIT	S2	S0, S1	None
8	SELECT * FROM TBLE	S2	S0, S1	None
9	DISCONNECT S1	S2	S0	None
	SELECT * FROM TBLF	S2	S0	None

Related concepts:

- "Distributed relational databases" in the *SQL Reference, Volume 1*

Related reference:

- "CONNECT (Type 1)" on page 134

Related samples:

- "dbmcon.sqc -- How to use multiple databases (C)"
- "dbmcon.sqC -- How to use multiple databases (C++)"

CREATE ALIAS

The CREATE ALIAS statement defines an alias for a table, view, nickname, or another alias.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

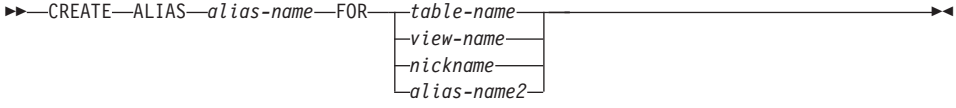
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the alias does not exist
- CREATEIN privilege on the schema, if the schema name of the alias refers to an existing schema.

To use the referenced object via the alias, the same privileges are required on that object as would be necessary if the object itself were used.

Syntax:



Description:

alias-name
Names the alias. The name must not identify a table, view, nickname, or alias that exists in the current database.

If a two-part name is specified, the schema name cannot begin with 'SYS' (SQLSTATE 42939).

The rules for defining an alias name are the same as those used for defining a table name.

FOR *table-name, view-name, nickname,* or *alias-name2*
Identifies the table, view, nickname, or alias for which *alias-name* is defined. If another alias name is supplied (*alias-name2*), then it must not

be the same as the new *alias-name* being defined (in its fully-qualified form). The *table-name* cannot be a declared temporary table (SQLSTATE 42995).

Notes:

- ***Compatibilities***
  - For compatibility with DB2 UDB for OS/390 and z/OS:
    - SYNONYM can be specified in place of ALIAS
  - The definition of the newly created alias is stored in SYSCAT.TABLES.
  - An alias can be defined for an object that does not exist at the time of the definition. If it does not exist, a warning is issued (SQLSTATE 01522). However, the referenced object must exist when a SQL statement containing the alias is compiled, otherwise an error is issued (SQLSTATE 52004).
  - An alias can be defined to refer to another alias as part of an alias chain but this chain is subject to the same restrictions as a single alias when used in an SQL statement. An alias chain is resolved in the same way as a single alias. If an alias used in a view definition, a statement in a package, or a trigger points to an alias chain, then a dependency is recorded for the view, package, or trigger on each alias in the chain. Repetitive cycles in an alias chain are not allowed and are detected at alias definition time.
  - Creating an alias with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

Examples:

Example 1: HEDGES attempts to create an alias for a table T1 (both unqualified).

```
CREATE ALIAS A1 FOR T1
```

The alias HEDGES.A1 is created for HEDGES.T1.

Example 2: HEDGES attempts to create an alias for a table (both qualified).

```
CREATE ALIAS HEDGES.A1 FOR MCKNIGHT.T1
```

The alias HEDGES.A1 is created for MCKNIGHT.T1.

Example 3: HEDGES attempts to create an alias for a table (alias in a different schema; HEDGES is not a DBADM; HEDGES does not have CREATEIN on schema MCKNIGHT).

```
CREATE ALIAS  MCKNIGHT.A1 FOR MCKNIGHT.T1
```

This example fails (SQLSTATE 42501).

Example 4: HEDGES attempts to create an alias for an undefined table (both qualified; FUZZY.WUZZY does not exist).

```
CREATE ALIAS HEDGES.A1 FOR FUZZY.WUZZY
```

This statement succeeds but with a warning (SQLSTATE 01522).

Example 5: HEDGES attempts to create an alias for an alias (both qualified).

```
CREATE ALIAS HEDGES.A1 FOR MCKNIGHT.T1
CREATE ALIAS HEDGES.A2 FOR HEDGES.A1
```

The first statement succeeds (as per example 2).

The second statement succeeds and an alias chain is created, consisting of HEDGES.A2 which refers to HEDGES.A1 which refers to MCKNIGHT.T1. Note that it does not matter whether or not HEDGES has any privileges on MCKNIGHT.T1. The alias is created regardless of the table privileges.

Example 6: Designate A1 as an alias for the nickname FUZZYBEAR.

```
CREATE ALIAS A1 FOR FUZZYBEAR
```

Example 7: A large organization has a finance department numbered D108 and a personnel department numbered D577. D108 keeps certain information in a table that resides at a DB2 RDBMS. D577 keeps certain records in a table that resides at an Oracle RDBMS. A DBA defines the two RDBMSs as data sources within a federated system, and gives the tables the nicknames of DEPTD108 and DEPTD577, respectively. A federated system user needs to create joins between these tables, but would like to reference them by names that are more meaningful than their alphanumeric nicknames. So the user defines FINANCE as an alias for DEPTD108 and PERSONNEL as an alias for DEPTD577.

```
CREATE ALIAS FINANCE FOR DEPTD108
CREATE ALIAS PERSONNEL FOR DEPTD577
```

CREATE BUFFERPOOL

The CREATE BUFFERPOOL statement creates a new buffer pool to be used by the database manager.

In a partitioned database, a default buffer pool definition is specified for each partition, with the capability to override the size on specific partitions. Also, in a partitioned database, the buffer pool is defined on all partitions unless database partition groups are specified. If database partition groups are specified, the buffer pool will only be created on partitions that are in those database partition groups.

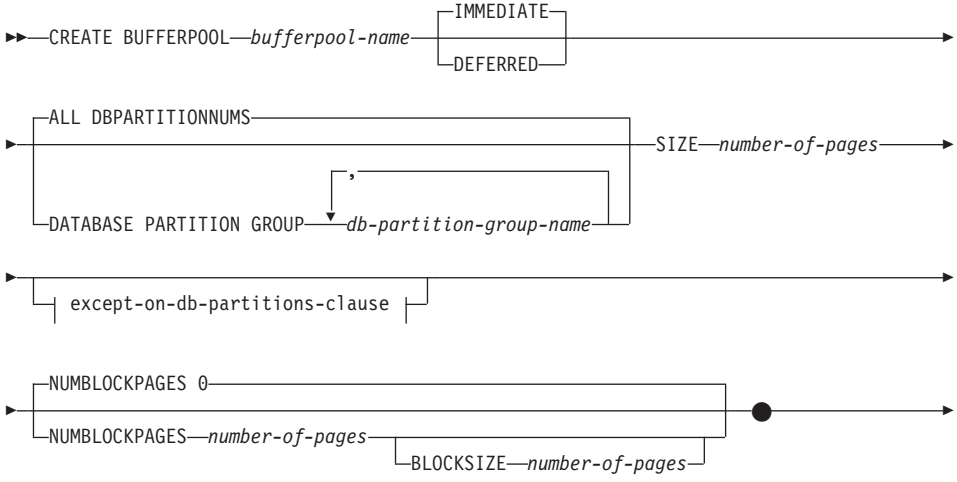
Invocation:

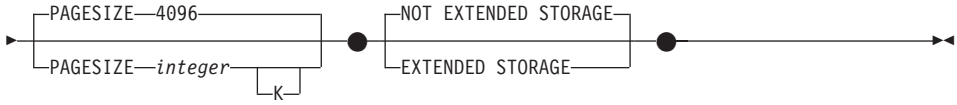
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

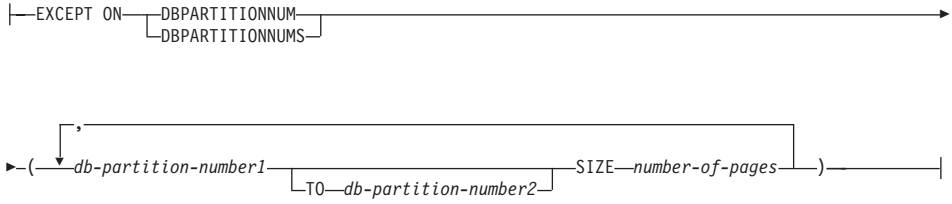
The authorization ID of the statement must have SYSCTRL or SYSADM authority.

Syntax:





except-on-db-partitions-clause:



Description:

bufferpool-name
Names the buffer pool. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *bufferpool-name* must not identify a buffer pool that already exists in a catalog (SQLSTATE 42710). The *bufferpool-name* must not begin with the characters 'SYS' and 'IBM' (SQLSTATE 42939).

IMMEDIATE
The bufferpool will be created immediately. If there is not enough reserved space in the database shared memory to allocate the new buffer pool, a warning (SQLSTATE 01657) is returned, and the statement is executed DEFERRED.

DEFERRED
The bufferpool will be created when the database is deactivated (all applications need to be disconnected from the database). Reserved memory space is not needed; DB2 will allocate the required memory from the system.

ALL DBPARTITIONNUMS
This buffer pool will be created on all partitions in the database.

DATABASE PARTITION GROUP *db-partition-group-name, ...*
Identifies the database partition group or groups to which the buffer pool definition applies. If this is specified, this buffer pool will only be created on partitions in these database partition groups. Each database partition group must currently exist in the database (SQLSTATE 42704). If the DATABASE PARTITION GROUP keywords are not specified, then this buffer pool will be created on all partitions (and any partitions subsequently added to the database).

SIZE *number-of-pages*
The size of the buffer pool specified as the number of pages. In a partitioned database, this will be the default size for all partitions where the buffer pool exists.

except-on-db-partitions-clause
Specifies the partition or partitions for which the size of the buffer pool will be different than the default. If this clause is not specified, then all partitions will have the same size as specified for this buffer pool.

EXCEPT ON DBPARTITIONNUMS
Keywords that indicate that specific partitions are specified. DBPARTITIONNUM is a synonym for DBPARTITIONNUMS.

db-partition-number1
Specifies a specific partition number that is included in the partitions for which the buffer pool is created.

TO *db-partition-number2*
Specify a range of partition numbers. The value of *db-partition-number2* must be greater than or equal to the value of *db-partition-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers must be included in the partitions for which the buffer pool is created (SQLSTATE 42729).

SIZE *number-of-pages*
The size of the buffer pool specified as the number of pages.

NUMBLOCKPAGES *number-of-pages*
Specifies the number of pages that should exist in the block-based area. The number of pages must not be greater than 98 percent of the number of pages for the buffer pool (SQLSTATE 54052). Specifying the value 0 disables block I/O. The actual value of NUMBLOCKPAGES used will be a multiple of BLOCKSIZE.

BLOCKSIZE *number-of-pages*
Specifies the number of pages in a block. The block size must be a value between 2 and 256 (SQLSTATE 54053). The default value is 32.

PAGESIZE *integer* **[K]**
Defines the size of pages used for the bufferpool. The valid values for *integer* without the suffix K are 4 096, 8 192, 16 384 or 32 768. The valid values for *integer* with the suffix K are 4, 8, 16 or 32. An error occurs if the page size is not one of these values (SQLSTATE 428DE). The default is 4 096 byte (4K) pages. Any number of spaces is allowed between *integer* and K, including no space.

EXTENDED STORAGE
If extended storage is enabled, pages that are being evicted from this

buffer pool will be cached in extended storage. (Extended storage is enabled by setting the database configuration parameters NUM_ESTORE_SEGS and ESTORE_SEG_SIZE to non-zero values.)

NOT EXTENDED STORAGE
Even if extended storage is enabled, pages that are being evicted from this buffer pool are not cached in extended storage.

Notes:

- *Compatibilities*
  - For compatibility with previous versions of DB2:
    - NODE can be specified in place of DBPARTITIONNUM
    - NODES can be specified in place of DBPARTITIONNUMS
    - NODEGROUP can be specified in place of DATABASE PARTITION GROUP
- If the buffer pool is created using the DEFERRED option, any table space created in this buffer pool will use a small system buffer pool of the same page size, until next database activation. The database has to be restarted for the buffer pool to become active and for table space assignments to the new buffer pool to take effect. The default option is IMMEDIATE.
- A buffer pool cannot be created with both extended storage and block-based support.
- There should be enough real memory on the machine for the total of all the buffer pools, as well as for the rest of the database manager and application requirements. If DB2 is unable to obtain memory for the regular buffer pools, it will attempt to start up with small system buffer pools, one for each page size (4K, 8K, 16K and 32K). In this situation, a warning will be returned to the user (SQLSTATE 01626), and the pages from all table spaces will use the system buffer pools.

Related reference:

- "Database Shared Memory Size configuration parameter - database_memory" in the *Administration Guide: Performance*

Related samples:

- "tscreate.sqc -- How to create and drop buffer pools and table spaces (C)"
- "tscreate.sqC -- How to create and drop buffer pools and table spaces (C++)"

CREATE DATABASE PARTITION GROUP

The CREATE DATABASE PARTITION GROUP statement creates a new database partition group within the database, assigns partitions to the database partition group, and records the database partition group definition in the catalog.

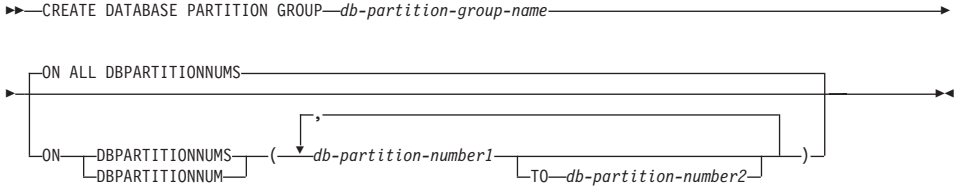
Invocation:

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement must have SYSCTRL or SYSADM or authority.

Syntax:



Description:

db-partition-group-name
Names the database partition group. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *db-partition-group-name* must not identify a database partition group that already exists in the catalog (SQLSTATE 42710). The *db-partition-group-name* must not begin with the characters 'SYS' or 'IBM' (SQLSTATE 42939).

ON ALL DBPARTITIONNUMS
Specifies that the database partition group is defined over all partitions defined to the database (`db2nodes.cfg` file) at the time the database partition group is created.

If a partition is added to the database system, the ALTER DATABASE PARTITION GROUP statement should be issued to include this new partition in a database partition group (including IBMDEFAULTGROUP). Furthermore, the REDISTRIBUTE DATABASE PARTITION GROUP command must be issued to move data to the partition.

ON DBPARTITIONNUMS
Specifies the specific partitions that are in the database partition group. DBPARTITIONNUM is a synonym for DBPARTITIONNUMS.

db-partition-number1
Specify a specific partition number. (A *node-name* of the form `NODEnnnnn` `can be specified for compatibility with the previous version.`)

TO *db-partition-number2*
Specify a range of partition numbers. The value of *db-partition-number2* must be greater than or equal to the value of *db-partition-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers are included in the database partition group.

Rules:

- Each partition specified by number must be defined in the `db2nodes.cfg` file (SQLSTATE 42729).
- Each *db-partition-number* listed in the ON DBPARTITIONNUMS clause must be appear at most once (SQLSTATE 42728).
- A valid *db-partition-number* is between 0 and 999 inclusive (SQLSTATE 42729).

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NODE can be specified in place of DBPARTITIONNUM
    - NODES can be specified in place of DBPARTITIONNUMS
    - NODEGROUP can be specified in place of DATABASE PARTITION GROUP
- This statement creates a partitioning map for the database partition group. A partitioning map identifier (PMAP_ID) is generated for each partitioning map. This information is recorded in the catalog and can be retrieved from SYSCAT.DBPARTITIONGROUPS and SYSCAT.PARTITIONMAPS. Each entry in the partitioning map specifies the target partition on which all rows that are hashed reside. For a single-partition database partition group, the corresponding partitioning map has only one entry. For a multiple partition database partition group, the corresponding partitioning map has 4 096 entries, where the partition numbers are assigned to the map entries in a round-robin fashion, by default.

Examples:

Assume that you have a partitioned database with six partitions defined as: 0, 1, 2, 5, 7, and 8.

- Assume that you want to create a database partition group called MAXGROUP on all six partitions. The statement is as follows:

```
CREATE DATABASE PARTITION GROUP MAXGROUP ON ALL DBPARTITIONNUMS
```

- Assume that you want to create a database partition group called MEDGROUP on partitions 0, 1, 2, 5, and 8. The statement is as follows:

```
CREATE DATABASE PARTITION GROUP MEDGROUP
  ON DBPARTITIONNUMS( 0 TO 2, 5, 8)
```

- Assume that you want to create a single-partition database partition group MINGROUP on partition 7. The statement is as follows:

```
CREATE DATABASE PARTITION GROUP MINGROUP
  ON DBPARTITIONNUM (7)
```

Related concepts:

- "Data partitioning across multiple partitions" in the *SQL Reference, Volume 1*

CREATE DISTINCT TYPE

The CREATE DISTINCT TYPE statement defines a distinct type. The distinct type is always sourced on one of the built-in data types. Successful execution of the statement also generates functions to cast between the distinct type and its source type and, optionally, generates support for the comparison operators (=, <>, <, <=, >, and >=) for use with the distinct type.

Invocation:

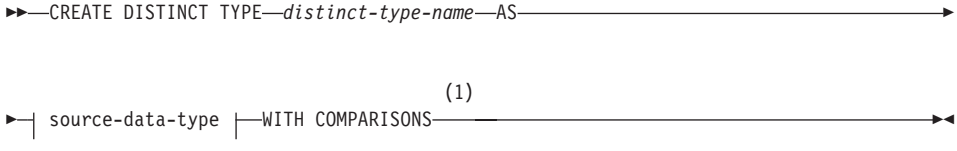
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

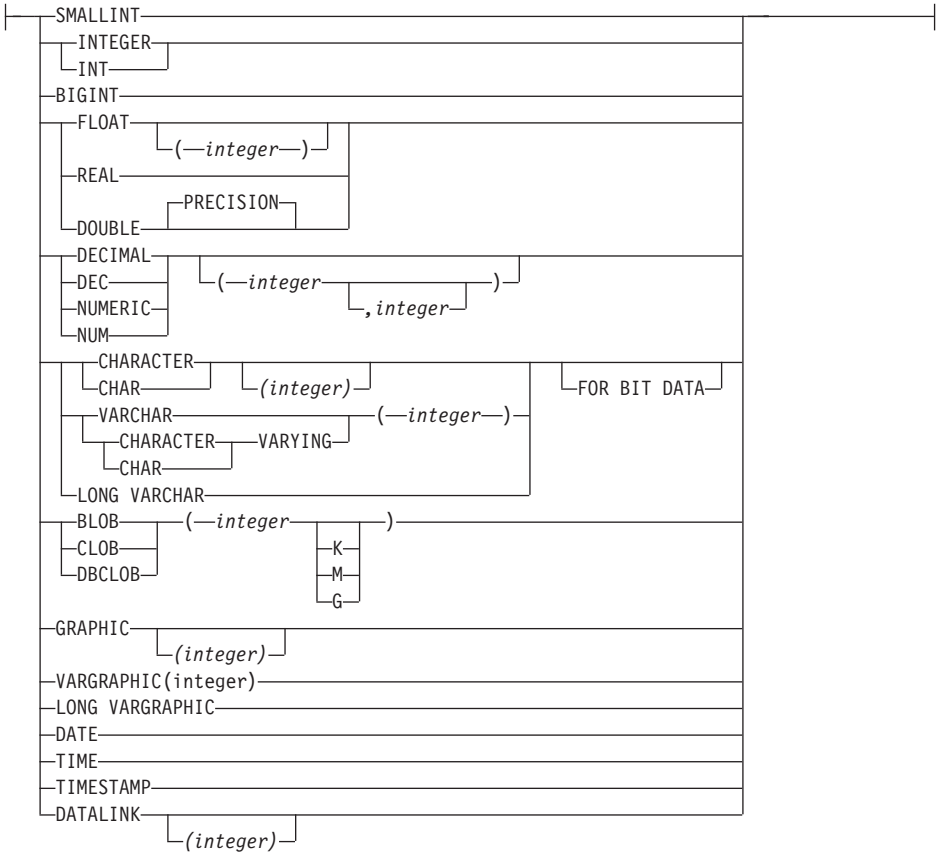
The privileges held by the authorization ID of the statement must include as least one of the following:

- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the database, if the schema name of the distinct type does not refer to an existing schema.
- CREATEIN privilege on the schema, if the schema name of the distinct type refers to an existing schema.

Syntax:



source-data-type:



Notes:

1 Required for all source-data-types except LOBs, LONG VARCHAR, LONG VARGRAPHIC and DATALINK which are not supported.

Description:

distinct-type-name
Names the distinct type. The name, including the implicit or explicit qualifier must not identify a distinct type described in the catalog. The unqualified name must not be the same as the name of a source-data-type or BOOLEAN (SQLSTATE 42918).

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The schema name (implicit or explicit) must not be greater than 8 bytes (SQLSTATE 42622).

A number of names used as keywords in predicates are reserved for system use, and may not be used as a *distinct-type-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators. Failure to observe this rule will lead to an error (SQLSTATE 42939).

If a two-part *distinct-type-name* is specified, the schema name cannot begin with 'SYS'; otherwise, an error (SQLSTATE 42939) is raised.

source-data-type
Specifies the data type used as the basis for the internal representation of the distinct type.

WITH COMPARISONS
Specifies that system-generated comparison operators are to be created for comparing two instances of a distinct type. These keywords should not be specified if the source-data-type is BLOB, CLOB, DBCLOB, LONG VARCHAR, LONG VARGRAPHIC, or DATALINK, otherwise a warning will be returned (SQLSTATE 01596) and the comparison operators will not be generated. For all other source-data-types, the WITH COMPARISONS keywords are required.

Notes:

- ***Privileges***

  The definer of the user-defined type always receives the EXECUTE privilege WITH GRANT OPTION on all functions automatically generated for the distinct type.

  EXECUTE privilege on all functions automatically generated during the CREATE DISTINCT TYPE is granted to PUBLIC.
- Creating a distinct type with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- The following functions are generated to cast to and from the source type:
  - One function to convert from the distinct type to the source type
  - One function to convert from the source type to the distinct type
  - One function to convert from INTEGER to the distinct type if the source type is SMALLINT
  - one function to convert from VARCHAR to the distinct type if the source type is CHAR
  - one function to convert from VARGRAPHIC to the distinct type if the source type is GRAPHIC.

In general these functions will have the following format:

```
CREATE FUNCTION source-type-name (distinct-type-name)
  RETURNS source-type-name ...

CREATE FUNCTION distinct-type-name (source-type-name)
  RETURNS distinct-type-name ...
```

In cases in which the source type is a parameterized type, the function to convert from the distinct type to the source type will have as function name the name of the source type without the parameters (see Table 3 for details). The type of the return value of this function will include the parameters given on the CREATE DISTINCT TYPE statement. The function to convert from the source type to the distinct type will have an input parameter whose type is the source type including its parameters. For example,

```
CREATE DISTINCT TYPE T_SHOESIZE AS CHAR(2)
  WITH COMPARISONS

CREATE DISTINCT TYPE T_MILES AS DOUBLE
  WITH COMPARISONS
```

will generate the following functions:

```
FUNCTION CHAR (T_SHOESIZE) RETURNS CHAR (2)

FUNCTION T_SHOESIZE (CHAR (2))
RETURNS T_SHOESIZE

FUNCTION DOUBLE (T_MILES) RETURNS DOUBLE

FUNCTION T_MILES (DOUBLE) RETURNS T_MILES
```

The schema of the generated cast functions is the same as the schema of the distinct type. No other function with this name and with the same signature may already exist in the database (SQLSTATE 42710).

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 3. CAST functions on distinct types

Source Type Name	Function Name	Parameter	Return-type
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*)

Table 3. CAST functions on distinct types (continued)

Source Type Name	Function Name	Parameter	Return-type
LONG VARCHAR	*distinct-type-name*	LONG VARCHAR	*distinct-type-name*
	LONG_VARCHAR	*distinct-type-name*	LONG VARCHAR
CLOB	*distinct-type-name*	CLOB (*n*)	*distinct-type-name*
	CLOB	*distinct-type-name*	CLOB (*n*)
BLOB	*distinct-type-name*	BLOB (*n*)	*distinct-type-name*
	BLOB	*distinct-type-name*	BLOB (*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*)
LONG VARGRAPHIC	*distinct-type-name*	LONG VARGRAPHIC	*distinct-type-name*
	LONG_VARGRAPHIC	*distinct-type-name*	LONG VARGRAPHIC
DBCLOB	*distinct-type-name*	DBCLOB (*n*)	*distinct-type-name*
	DBCLOB	*distinct-type-name*	DBCLOB (*n*)
SMALLINT	*distinct-type-name*	SMALLINT	*distinct-type-name*
	distinct-type-name	INTEGER	*distinct-type-name*
	SMALLINT	*distinct-type-name*	SMALLINT
INTEGER	*distinct-type-name*	INTEGER	*distinct-type-name*
	INTEGER	*distinct-type-name*	INTEGER
BIGINT	*distinct-type-name*	BIGINT	*distinct-type-name*
	BIGINT	*distinct-type-name*	BIGINT
DECIMAL	*distinct-type-name*	DECIMAL (*p,s*)	*distinct-type-name*
	DECIMAL	*distinct-type-name*	DECIMAL (*p,s*)
NUMERIC	*distinct-type-name*	DECIMAL (*p,s*)	*distinct-type-name*
	DECIMAL	*distinct-type-name*	DECIMAL (*p,s*)
REAL	*distinct-type-name*	REAL	*distinct-type-name*
	distinct-type-name	DOUBLE	*distinct-type-name*
	REAL	*distinct-type-name*	REAL
FLOAT(*n*) where *n*<=24	*distinct-type-name*	REAL	*distinct-type-name*
	distinct-type-name	DOUBLE	*distinct-type-name*
	REAL	*distinct-type-name*	REAL

Table 3. CAST functions on distinct types (continued)

Source Type Name	Function Name	Parameter	Return-type
FLOAT(*n*) where *n*>24	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
FLOAT	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
DOUBLE	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
DOUBLE PRECISION	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
DATE	*distinct-type-name*	DATE	*distinct-type-name*
	DATE	*distinct-type-name*	DATE
TIME	*distinct-type-name*	TIME	*distinct-type-name*
	TIME	*distinct-type-name*	TIME
TIMESTAMP	*distinct-type-name*	TIMESTAMP	*distinct-type-name*
	TIMESTAMP	*distinct-type-name*	TIMESTAMP
DATALINK	*distinct-type-name*	DATALINK	*distinct-type-name*
	DATALINK	*distinct-type-name*	DATALINK

Note: NUMERIC and FLOAT are not recommended when creating a user-defined type for a portable application. DECIMAL and DOUBLE should be used instead.

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, etc.) are supported on distinct types until the CREATE FUNCTION statement is used to register user-defined functions for the distinct type, where those user-defined functions are sourced on the appropriate built-in functions. In particular, note that it is possible to register user-defined functions that are sourced on the built-in column functions.

When a distinct type is created using the WITH COMPARISONS clause, system-generated comparison operators are created. Creation of these comparison operators will generate entries in the SYSCAT.ROUTINES catalog view for the new functions.

The schema name of the distinct type must be included in the SQL path or the FUNCPATH BIND option for successful use of these operators and cast functions in SQL statements.

Examples:

Example 1: Create a distinct type named SHOESIZE that is based on an INTEGER data type.

```
CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS
```

This will also result in the creation of comparison operators (=, <>, <, <=, >, >=) and cast functions INTEGER(SHOESIZE) returning INTEGER and SHOESIZE(INTEGER) returning SHOESIZE.

Example 2: Create a distinct type named MILES that is based on a DOUBLE data type.

```
CREATE DISTINCT TYPE MILES AS DOUBLE WITH COMPARISONS
```

This will also result in the creation of comparison operators (=, <>, <, =, >, >=) and cast functions DOUBLE(MILES) returning DOUBLE and MILES(DOUBLE) returning MILES.

Related reference:

- "Basic predicate" in the *SQL Reference, Volume 1*
- "CREATE FUNCTION" on page 188
- "CREATE TABLE" on page 332
- "SET PATH" on page 726
- "User-defined types" in the *SQL Reference, Volume 1*

Related samples:

- "dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)"
- "udfcli.sqc -- Call a variety of types of user-defined functions (C)"
- "dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)"
- "udfcli.sqC -- Call a variety of types of user-defined functions (C++)"
- "DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)"
- "DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)"

CREATE EVENT MONITOR

The CREATE EVENT MONITOR statement defines a monitor that will record certain events that occur when using the database. The definition of each event monitor also specifies where the database should record the events.

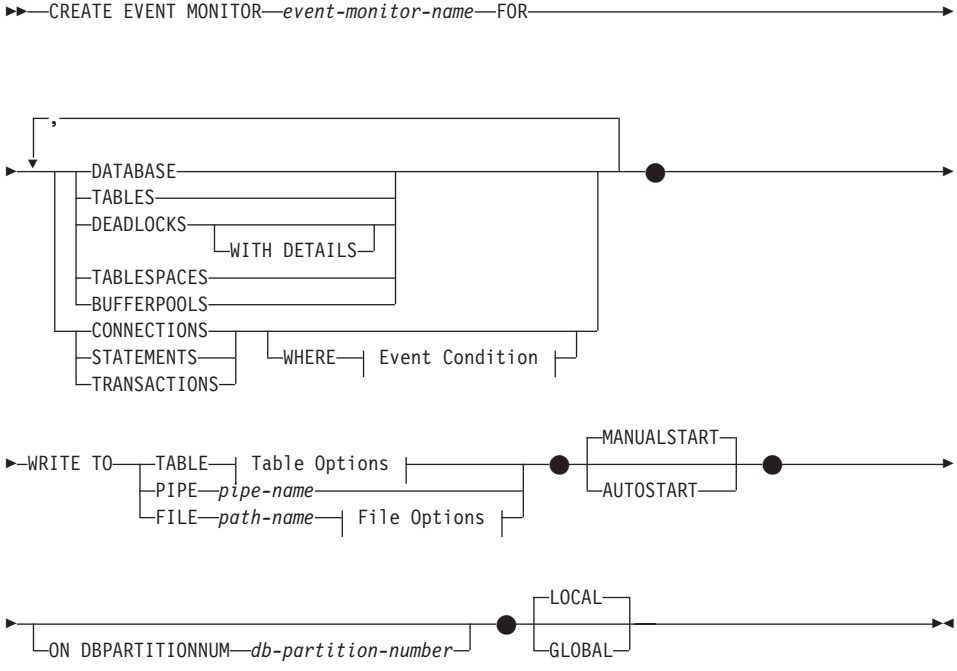
Invocation:

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID must include either SYSADM or DBADM authority (SQLSTATE 42502).

Syntax:



Event Condition:

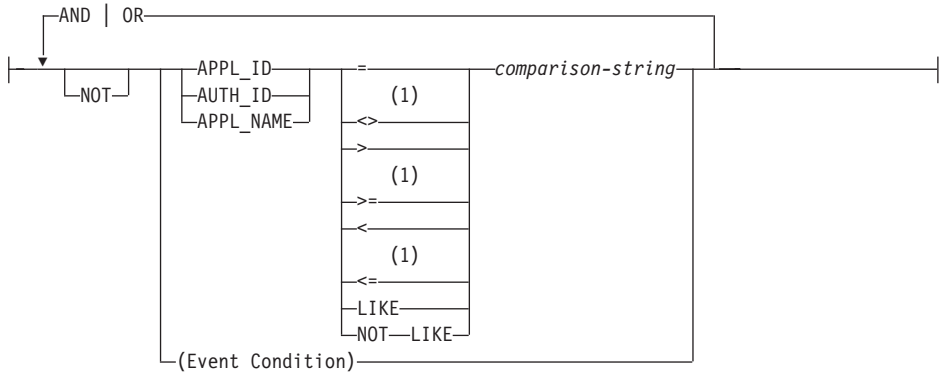
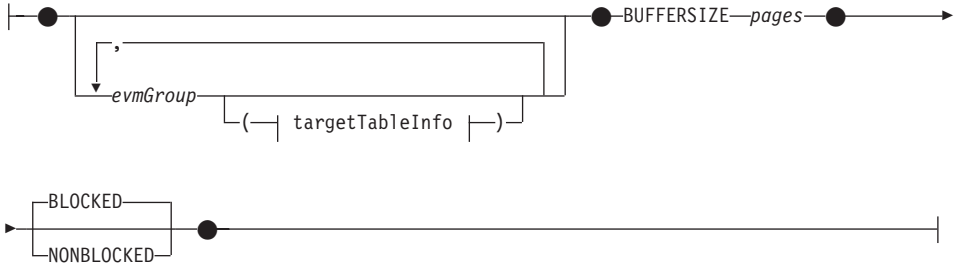
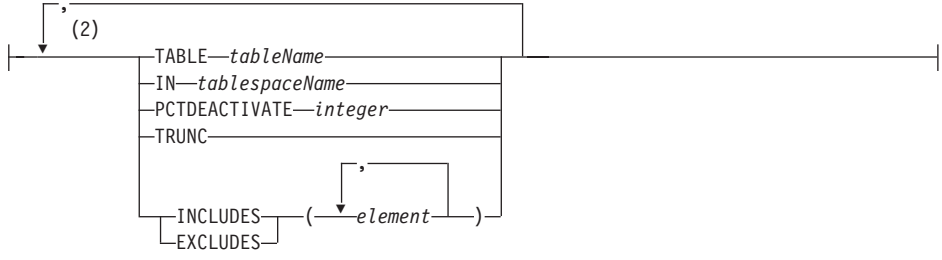


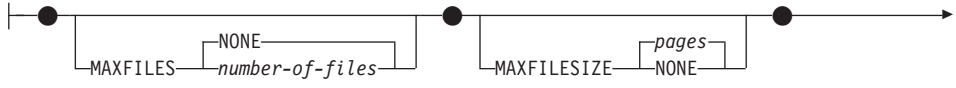
Table Options:

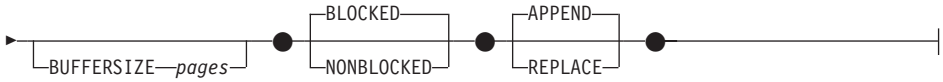


targetTableInfo:



File Options:





Notes:

1 Other forms of these operators are also supported.

2 Each clause may be specified only once.

Description:

event-monitor-name
Names the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that already exists in the catalog (SQLSTATE 42710).

FOR
Introduces the type of event to record.

DATABASE
Specifies that the event monitor records a database event when the last application disconnects from the database.

TABLES
Specifies that the event monitor records a table event for each active table when the last application disconnects from the database. An active table is a table that has changed since the first connection to the database.

DEADLOCKS
Specifies that the event monitor records a deadlock event whenever a deadlock occurs. Specifying the WITH DETAILS option indicates that the event monitor will generate a more detailed deadlock connection event for each application involved in a deadlock. This additional detail includes:

- Information about the statement that the application was executing when the deadlock occurred, such as the statement text.
- The locks held by the application when the deadlock occurred. In a partitioned database environment, the locks included are only those on the database partition where the application was waiting for its lock when the deadlock occurred.

Both DEADLOCKS and DEADLOCKS WITH DETAILS cannot be specified in the same statement (SQLSTATE 42613).

TABLESPACES
Specifies that the event monitor records a table space event for each table space when the last application disconnects from the database.

BUFFERPOOLS
Specifies that the event monitor records a buffer pool event when the last application disconnects from the database.

CONNECTIONS
Specifies that the event monitor records a connection event when an application disconnects from the database.

STATEMENTS
Specifies that the event monitor records a statement event whenever a SQL statement finishes executing.

TRANSACTIONS
Specifies that the event monitor records a transaction event whenever a transaction completes (that is, whenever there is a commit or rollback operation).

WHERE *event condition*
Defines a filter that determines which connections cause a CONNECTION, STATEMENT or TRANSACTION event to occur. If the result of the event condition is TRUE for a particular connection, then that connection will generate the requested events.

This clause is a special form of the WHERE clause that should not be confused with a standard search condition.

To determine if an application will generate events for a particular event monitor, the WHERE clause is evaluated:

1. For each active connection when an event monitor is first turned on.
2. Subsequently for each new connection to the database at connect time.

The WHERE clause is not evaluated for each event.

If no WHERE clause is specified then all events of the specified event type will be monitored.

APPL_ID
Specifies that the application ID of each connection should be compared with the *comparison-string* in order to determine if the connection should generate CONNECTION, STATEMENT or TRANSACTION events (whichever was specified).

AUTH_ID
Specifies that the authorization ID of each connection should be compared with the *comparison-string* in order to determine if the connection should generate CONNECTION, STATEMENT or TRANSACTION events (whichever was specified).

APPL_NAME
Specifies that the application program name of each connection should be compared with the *comparison-string* in order to determine if the connection should generate CONNECTION, STATEMENT or TRANSACTION events (whichever was specified).

The application program name is the first 20 bytes of the application program file name, after the last path separator.

comparison-string
A string to be compared with the APPL_ID, AUTH_ID, or APPL_NAME of each application that connects to the database. *comparison-string* must be a string constant (that is, host variables and other string expressions are not permitted).

WRITE TO
Introduces the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of database tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Data for groups having a target table is kept, whereas data for groups not having a target table is discarded. Each monitor element contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table. Other elements are discarded.

Table Options
Specifies table formatting options.

evmGroupInfo
Defines the target table for a logical data group. This clause should be specified for each grouping that is to be recorded. However, if no evmGroupInfo clauses are specified, all groups for the event monitor type are recorded.

evmGroup
Identifies the logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Type of Event Monitor	evmGroup Value
Database	DB
	CONTROL[1]
Tables	TABLE
	CONTROL[1]

Type of Event Monitor	evmGroup Value
Deadlocks	CONNHEADER DEADLOCK DLCONN CONTROL[1]
Deadlocks with details	CONNHEADER DEADLOCK DLCONN[2] DLLOCK[3] CONTROL[1]
Tablespaces	TABLESPACE CONTROL[1]
Bufferpools	BUFFERPOOL CONTROL[1]
Connections	CONNHEADER CONN CONTROL[1]
Statements	CONNHEADER STMT SUBSECTION[4] CONTROL[1]
Transactions	CONNHEADER XACT CONTROL[1]

[1] Logical data groups dbheader (conn_time element only), start and overflow, are all written to the CONTROL group. Overflow is written if the event monitor is non-blocked and events were discarded.

[2] Corresponds to the DETAILED_DLCONN event.

[3] Corresponds to the LOCK logical data groups that occur within each DETAILED_DLCONN event.

[4] Created only for partitioned database environments.

targetTableInfo

Identifies the target table for the group. If a value for *targetTableInfo* is not specified, CREATE EVENT MONITOR processing proceeds as follows:

- A derived table name is used (described below).
- A default table space is chosen (described below).
- All elements are included.

- PCTDEACTIVATE and TRUNC are not specified.

TABLE *tableName*
Specifies the name of the target table. If the name is unqualified, the table schema defaults to the schema for the current authorization ID. If no name is provided, the unqualified name is derived from *evmGroup* and *event-monitor-name* as follows:

```
substring(evmGroup || "_" || event-monitor-name,
  1,128)
```

IN *tablespaceName*
Defines the table space in which the table is to be created. If no table space name is provided, the table space is chosen as follows:

```
IF table space IBMDEFAULTGROUP over which the user
 has USE privilege exists
  THEN choose it
ELSE IF a table space over which the user
 has USE privilege exists
  THEN choose it
ELSE issue an error (SQLSTATE 42727)
```

PCTDEACTIVATE *integer*
If a table is being created in a DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100. The default value is 100 (meaning that the event monitor deactivates when the table space becomes completely full). This option cannot be specified with SMS table spaces.

TRUNC
Specifies that the stmt_text column is defined as VARCHAR(*n*), where *n* is the largest size that can fit into the table row. In this case, any statement text that is longer than *n* will be truncated. The following example illustrates how the value of *n* is calculated. Assume that:

- The stmt table is created in a table space that uses 32K pages.
- The total length of all the other columns in the table equals 357 bytes.

In this case, the maximum row size for a table is 32677 bytes. Therefore, stmt_text would be defined as VARCHAR(32317) (that is, 32644 - 357 - 4). If TRUNC

is not specified, the stmt_text column will be defined as CLOB(64K). Note that stmt_text is found in the STMT group and the DLCONN group (for deadlocks with details event monitors).

INCLUDES
Specifies that the following elements are to be included in the table.

EXCLUDES
Specifies that the following elements are *not* to be included in the table.

element
Identifies a monitor element. Element information can be provided in one of the following forms:

- Specify no element information. In this case, all elements are included in the CREATE TABLE statement.
- Specify the elements to include in the form: INCLUDES (element1, element2, ..., element*n*). Only table columns are created for these elements.
- Specify the elements to exclude in the form: EXCLUDES (element1, element2, ..., element*n*). Only table columns are created for all elements except these.

Use the db2evtbl command to build a CREATE EVENT MONITOR statement that includes a complete list of elements for a group.

BUFFERSIZE *pages*
Specifies the size of the event monitor buffers (in units of 4K pages). Table event monitors insert all data from a buffer, and issues a COMMIT once the buffer has been processed. The larger the buffers, the larger the commit scope used by the event monitor. Highly active event monitors should have larger buffers than relatively inactive event monitors. When a monitor is started, two buffers of the specified size are allocated. Event monitors use double buffering to permit asynchronous I/O.

The minimum (and default) size of each buffer is 4 pages (that is, 2 buffers, each 16K in size). The maximum size of the buffers is limited by the size of the monitor heap (MON_HEAP), because the buffers are allocated from that heap. If using many event monitors at the same time, increase the size of the MON_HEAP database configuration parameter.

BLOCKED
Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED should be selected to guarantee no event data loss. This is the default option.

NONBLOCKED
Specifies that each agent that generates an event should not wait for the event buffer to be written out to disk if the agent determines that both event buffers are full. NONBLOCKED event monitors do not slow down database operations to the extent of BLOCKED event monitors. However, NONBLOCKED event monitors are subject to data loss on highly active systems.

PIPE
Specifies that the target for the event monitor data is a named pipe. The event monitor writes the data to the pipe in a single stream (that is, as if it were a single, infinitely long file). When writing the data to a pipe, an event monitor does not perform blocked writes. If there is no room in the pipe buffer, then the event monitor will discard the data. It is the monitoring application's responsibility to read the data promptly if it wishes to ensure no data loss.

pipe-name
The name of the pipe (FIFO on AIX) to which the event monitor will write the data.

The naming rules for pipes are platform specific. On UNIX operating systems pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see *path-name* below). However, on Windows NT/2000, there is a special syntax for a pipe name. As a result, on Windows NT/2000 absolute pipe names are required.

The existence of the pipe will not be checked at event monitor creation time. It is the responsibility of the monitoring application to have created and opened the pipe for reading at the time that the event monitor is activated. If the pipe is not available at this time, then the event monitor will turn itself off, and will log an error. (That is, if the event monitor was activated at database start time as a result of the AUTOSTART option, then the event monitor will log an error in the system error log.) If the event monitor is activated via the SET EVENT MONITOR STATE SQL statement, then that statement will fail (SQLSTATE 58030).

FILE
Indicates that the target for the event monitor data is a file (or set of files). The event monitor writes out the stream of data as a series of 8 character numbered files, with the extension "evt". (for example,

00000000.evt, 00000001.evt, and 00000002.evt). The data should be considered to be one logical file even though the data is broken up into smaller pieces (that is, the start of the data stream is the first byte in the file 00000000.evt; the end of the data stream is the last byte in the file nnnnnnnn.evt).

The maximum size of each file can be defined as well as the maximum number of files. An event monitor will never split a single event record across two files. However, an event monitor may write related records in two different files. It is the responsibility of the application that uses this data to keep track of such related information when processing the event files.

path-name
The name of the directory in which the event monitor should write the event files data. The path must be known at the server, however, the path itself could reside on another partition (for example, in a UNIX-based system, this might be an NFS mounted file). A string constant must be used when specifying the *path-name*.

The directory does not have to exist at CREATE EVENT MONITOR time. However, a check is made for the existence of the target path when the event monitor is activated. At that time, if the target path does not exist, an error (SQLSTATE 428A3) is raised.

If an absolute path (a path that starts with the root directory on AIX, or a disk identifier on Windows NT/2000) is specified, then the specified path will be the one used. If a relative path (a path that does not start with the root) is specified, then the path relative to the DB2EVENT directory in the database directory will be used.

When a relative path is specified, the DB2EVENT directory is used to convert it into an absolute path. Thereafter, no distinction is made between absolute and relative paths. The absolute path is stored in the SYSCAT.EVENTMONITORS catalog view.

It is possible to specify two or more event monitors that have the same target path. However, once one of the event monitors has been activated for the first time, and as long as the target directory is not empty, it will be impossible to activate any of the other event monitors.

File Options
Specifies the options for the file format.

MAXFILES NONE
Specifies that there is no limit to the number of event files that the event monitor will create. This is the default.

MAXFILES *number-of-files*
Specifies that there is a limit on the number of event monitor files that will exist for a particular event monitor at any time. Whenever an event monitor has to create another file, it will check to make sure that the number of .evt files in the directory is less than *number-of-files*. If this limit has already been reached, then the event monitor will turn itself off.

If an application removes the event files from the directory after they have been written, then the total number of files that an event monitor can produce can exceed *number-of-files*. This option has been provided to allow a user to guarantee that the event data will not consume more than a specified amount of disk space.

MAXFILESIZE *pages*
Specifies that there is a limit to the size of each event monitor file. Whenever an event monitor writes a new event record to a file, it checks that the file will not grow to be greater than *pages* (in units of 4K pages). If the resulting file would be too large, then the event monitor switches to the next file. The default for this option is:

- Windows NT/2000 - 200 4K pages
- UNIX - 1000 4K pages

The number of pages must be greater than at least the size of the event buffer in pages. If this requirement is not met, then an error (SQLSTATE 428A4) is raised.

MAXFILESIZE NONE
Specifies that there is no set limit on a file's size. If MAXFILESIZE NONE is specified, then MAXFILES 1 must also be specified. This option means that one file will contain all of the event data for a particular event monitor. In this case the only event file will be 00000000.evt.

BUFFERSIZE *pages*
Specifies the size of the event monitor buffers (in units of 4K pages). All event monitor file I/O is buffered to improve the performance of the event monitors. The larger the buffers, the less I/O will be performed by the event monitor. Highly active event monitors should have larger buffers than relatively inactive event monitors. When the monitor is

started, two buffers of the specified size are allocated. Event monitors use double buffering to permit asynchronous I/O.

The minimum and default size of each buffer (if this option is not specified) is 4 pages (that is, 2 buffers, each 16 K in size). The maximum size of the buffers is limited by the size of the monitor heap (MON_HEAP) since the buffers are allocated from the heap. If using a lot of event monitors at the same time, increase the size of the MON_HEAP database configuration parameter.

Event monitors that write their data to a pipe also have two internal (non-configurable) buffers that are each 1 page in size. These buffers are also allocated from the monitor heap (MON_HEAP). For each active event monitor that has a pipe target, increase the size of the database heap by 2 pages.

BLOCKED
Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED should be selected to guarantee no event data loss. This is the default option.

NONBLOCKED
Specifies that each agent that generates an event should not wait for the event buffer to be written out to disk if the agent determines that both event buffers are full. NONBLOCKED event monitors do not slow down database operations to the extent of BLOCKED event monitors. However, NONBLOCKED event monitors are subject to data loss on highly active systems.

APPEND
Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will append the new event data to the existing stream of data files. When the event monitor is reactivated, it will resume writing to the event files as if it had never been turned off. APPEND is the default option.

The APPEND option does not apply at CREATE EVENT MONITOR time, if there is existing event data in the directory where the newly created event monitor is to write its event data.

REPLACE
Specifies that if event data files already exist when the event

monitor is turned on, then the event monitor will erase all of the event files and start writing data to file 00000000.evt.

MANUALSTART
Specifies that the event monitor not be started automatically each time the database is started. Event monitors with the MANUALSTART option must be activated manually using the SET EVENT MONITOR STATE statement. This is the default option.

AUTOSTART
Specifies that the event monitor be started automatically each time the database is started.

ON DBPARTITIONNUM
Keyword that indicates that a specific database partition is specified.

db-partition-number
Specifies a database partition number where the event monitor runs and writes the events. With the monitoring scope defined as GLOBAL, all database partitions report to the specified database partition number. The I/O component will physically run on the specified database partition, writing its records to the file or pipe specified above.

GLOBAL
The event monitor reports on all database partitions. For a partitioned database in DB2 Universal Database Version 8, only deadlocks and deadlocks with details event monitors can be defined as GLOBAL.

LOCAL
The event monitor reports only on the database partition that is running. It gives a partial trace of the database activity. This is the default.

Rules:

- Each of the event types (DATABASE, TABLES, DEADLOCKs,...) can only be specified once in a particular event monitor definition.

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NODE can be specified in place of DBPARTITIONNUM
- Event monitor definitions are recorded in the SYSCAT.EVENTMONITORS catalog view. The events themselves are recorded in the SYSCAT.EVENTS catalog view. The names of target tables are recorded in the SYSCAT.EVENTTABLES catalog view.

- There is a performance impact when using DEADLOCKS WITH DETAILS rather than DEADLOCKS. When a deadlock occurs, the database manager requires extra time to record the extra deadlock information.
- A CONNHEADER event is normally written whenever a connection is established. However, if an event monitor is created only for DEADLOCKS WITH DETAILS, a CONNHEADER event will only be written the first time that the connection participates in a deadlock.
- The BUFFERSIZE parameter restricts the size of STMT and DETAILED_DLCONN events. If a STMT event cannot fit within a buffer, it is truncated by truncating statement text. If a DETAILED_DLCONN event cannot fit within a buffer, it is truncated by removing locks. If it still cannot fit, statement text is truncated.
- ***Write to table event monitors:***
  - General Notes:
    - All target tables are created when the CREATE EVENT MONITOR statement executes.
    - If the creation of a table fails for any reason, an error is passed back to the application program, and the CREATE EVENT MONITOR statement fails.
    - A target table can only be used by one event monitor. During CREATE EVENT MONITOR processing, if a target table is found to have already been defined for use by another event monitor, the CREATE EVENT MONITOR statement fails, and an error is passed back to the application program. A table is defined for use by another event monitor if the table name matches a value found in the SYSCAT.EVENTTABLES catalog view.
    - During CREATE EVENT MONITOR processing, if a table already exists, but is *not* defined for use by another event monitor, no table is created, and processing continues. A warning is passed back to the application program.
    - Any table spaces must exist before the CREATE EVENT MONITOR statement is executed. The CREATE EVENT MONITOR statement does not create table spaces.
    - If specified, the LOCAL and GLOBAL keywords are ignored. With WRITE TO TABLE event monitors, an event monitor output process or thread is started on each database partition in the instance, and each of these processes reports data only for the database partition on which it is running.
    - The following event types from the flat monitor log file or pipe format are not recorded by write to table event monitors:
      - LOG_STREAM_HEADER
      - LOG_HEADER

- DB_HEADER (Elements db_name and db_path are not recorded. The element conn_time is recorded in CONTROL.)

- In a partitioned database environment, the table space within which tables are defined must exist across all partitions that will have event monitor data written to them. Failure to observe this rule will result in records not being written to the log on partitions (with event monitors) where the table space does not exist. Events will still be written on partitions where the table space *does* exist, and no error will be returned. This behavior allows users to choose a subset of partitions for monitoring, by creating a table space that exists only on certain partitions.
- Users must manually prune all target tables.

– Table Columns:

- Column names in a table match an event monitor data element identifier. Monitor variables of type sqlm_time (elapsed time) are an exception. The column names for such types are type_name_s, and type_name_ms, representing the columns that store the time in seconds and microseconds, respectively. Any event monitor element that does *not* have a corresponding target table column is ignored.
- Use the db2evtbl command to build a CREATE EVENT MONITOR command that includes a complete list of elements for a group.
- The types of columns being used for monitor elements correlate to the following mapping:

```
SQLM_TYPE_STRING                    CHAR[n], VARCHAR[n] or CLOB(n)
                                    (If the data in the event monitor
                                    record exceeds n bytes,
                                    it is truncated.)
SQLM_TYPE_U8BIT and SQLM_TYPE_8BIT  SMALLINT, INTEGER or BIGINT
SQLM_TYPE_16BIT and SQLM_TYPE_U16BIT SMALLINT, INTEGER or BIGINT
SQLM_TYPE_32BIT and SQLM_TYPE_U32BIT INTEGER or BIGINT
SQLM_TYPE_U64BIT and SQLM_TYPE_64BIT BIGINT
sqlm_timestamp                      TIMESTAMP
sqlm_time(elapsed time)             BIGINT
sqlca:
  sqlerrmc                          VARCHAR[72]
  sqlstate                          CHAR[5]
  sqlwarn                           CHAR[11]
  other fields                      INTEGER or BIGINT
```

- Columns are defined to be NOT NULL.
- Because the performance of tables with CLOB columns is inferior to tables that have VARCHAR columns, consider using the TRUNC keyword when specifying the stmt evmGroup (or dlconn evmGroup, if using deadlocks with details).
- Unlike other target tables, the columns in the CONTROL table do not match data element identifiers. Columns are defined as follows:

Column Name	Data Type	Nullable	Description
PARTITION_KEY	INTEGER	N	Partition key (partitioned database only)
PARTITION_NUMBER	INTEGER	N	Partition number (partitioned database only)
EVMONNAME	VARCHAR(128)	N	Name of the event monitor
MESSAGE	VARCHAR(128)	N	Describes the nature of the MESSAGE_TIME column. This can be one of the following: - FIRST_CONNECT (the time of the first connect to the database after activation) - EVMON_START (the time that the event monitor listed in EVMONNAME was started) - OVERFLOWS:*n* (denotes that *n* records were discarded because of buffer overflow)
MESSAGE_TIME	TIMESTAMP	N	Timestamp

  - In a partitioned database environment, the first column of each table is named PARTITION_KEY, is NOT NULL, and is of type INTEGER. This column is used as the partitioning key for the table. The value of this column is chosen so that each event monitor process inserts data into the database partition on which the process is running; that is, insert operations are performed locally on the database partition where the event monitor process is running. On any database partition, the PARTITION_KEY field will contain the same value. This means that if a data partition is dropped and data redistribution is performed, all data on the dropped database partition will go to one other database partition instead of being evenly distributed. Therefore, before removing a database partition, consider deleting all table rows on that database partition.
  - In a partitioned database environment, a column named PARTITION_NUMBER can be defined for each table. This column is NOT NULL and is of type INTEGER. It contains the number of the partition on which the data was inserted. Unlike the PARTITION_KEY column, the PARTITION_NUMBER column is not mandatory. The PARTITION_NUMBER column is not allowed in a non-partitioned database environment.

– Table Attributes:
  - Default table attributes are used. Besides partitioning key (partitioned database only), no extra options are specified when creating tables.
  - Indexes on the table can be created.
  - Extra table attributes (such as volatile, RI, triggers, constraints, and so on) can be added, but the event monitor process (or thread) will ignore them.

- If "not logged initially" is added as a table attribute, it is turned off at the first COMMIT, and is not set back on.

– Event Monitor Activation:
  - When an event monitor activates, all target table names are retrieved from the SYSCAT.EVENTTABLES catalog view.
  - In a partitioned database environment, activation processing determines the table spaces and database partition groups in which the target tables reside. The event monitor only activates on those partitions that are included in the database partition group.
  - If a target table does not exist when the event monitor activates (or, in a partitioned database environment, if the table space does not reside on a database partition), activation continues, and data that would otherwise be inserted into this table is ignored.
  - Activation processing validates each target table. If validation fails, activation of the event monitor fails, and messages are written to the administration log.
  - During activation in a partitioned database environment, the CONTROL table rows for FIRST_CONNECT and EVMON_START are only inserted on the catalog database partition. This requires that the table space for the control table exist on the catalog database partition. If it does not exist on the catalog database partition, these inserts are not performed.
  - In a partitioned database environment, if a partition is not yet active when a write to table event monitor is activated, that partition is activated before the event monitor is activated. In this case, database activation behaves as if an SQL CONNECT statement has activated the database on all partitions.

– Run Time:
  - An event monitor runs with DBADM authority.
  - If, while an event monitor is active, an insert operation into a target table fails:
    • Uncommitted changes are rolled back.
    • A message is written to the administration log.
    • The event monitor is deactivated.
  - If an event monitor is active, it performs a local COMMIT when it has finished processing an event monitor buffer.
  - In a partitioned database environment, the actual statement text, which can be up to 65 535 bytes in length, is only stored (in the STMT or DLCONN table) by the event monitor process running on the application coordinator database partition. On other database partitions, this value has zero length.

- In a non-partitioned database environment, all write to table event monitors are deactivated when the last application terminates (and the database has not been explicitly activated). In a partitioned database environment, write to table event monitors are deactivated when the catalog partition deactivates.
- The DROP EVENT MONITOR statement does not drop target tables.

Examples:

Example 1: The following example creates an event monitor called SMITHPAY. This event monitor, will collect event data for the database as well as for the SQL statements performed by the PAYROLL application owned by the JSMITH authorization ID. The data will be appended to the absolute path /home/jsmith/event/smithpay/. A maximum of 25 files will be created. Each file will be a maximum of 1 024 4K pages long. The file I/O will be non-blocked.

```
CREATE EVENT MONITOR SMITHPAY
  FOR DATABASE, STATEMENTS
  WHERE APPL_NAME = 'PAYROLL' AND AUTH_ID = 'JSMITH'
  WRITE TO FILE '/home/jsmith/event/smithpay'
  MAXFILES 25
  MAXFILESIZE 1024
  NONBLOCKED
  APPEND
```

Example 2: The following example creates an event monitor called DEADLOCKS_EVTS. This event monitor will collect deadlock events and will write them to the relative path DLOCKS. One file will be written, and there is no maximum file size. Each time the event monitor is activated, it will append the event data to the file 00000000.evt if it exists. The event monitor will be started each time the database is started. The I/0 will be blocked by default.

```
CREATE EVENT MONITOR DEADLOCK_EVTS
  FOR DEADLOCKS
  WRITE TO FILE 'DLOCKS'
  MAXFILES 1
  MAXFILESIZE NONE
  AUTOSTART
```

Example 3: This example creates an event monitor called DB_APPLS. This event monitor collects connection events, and writes the data to the named pipe /home/jsmith/applpipe.

```
CREATE EVENT MONITOR DB_APPLS
  FOR CONNECTIONS
  WRITE TO PIPE '/home/jsmith/applpipe'
```

Example 4: This example, which assumes a partitioned database environment, creates an event monitor called FOO. This event monitor collects SQL statement events and writes them to SQL tables with the following derived names:

- CONNHEADER_FOO
- STMT_FOO
- SUBSECTION_FOO
- CONTROL_FOO

Because no table space information is supplied, all tables will be created in a table space selected by the system, based on the rules described under the IN *tablespaceName* clause. All tables include all elements for their group (that is, columns are defined whose names are equivalent to the element names.)

```
CREATE EVENT MONITOR FOO
  FOR STATEMENTS
  WRITE TO TABLE
```

Example 5: This example, which assumes a partitioned database environment, creates an event monitor called BAR. This event monitor collects SQL statement and transaction events and writes them to tables as follows:

- Any data from the STMT group is written to table MYDEPT.MYSTMTINFO. The table is created in table space MYTABLESPACE. Create columns only for the following elements: rows_read, rows_written, and stmt_text. Any other elements of the group will be discarded.
- Any data from the SUBSECTION group is written to table MYDEPT.MYSUBSECTIONINFO. The table is created in table space MYTABLESPACE. The table includes all columns, except start_time, stop_time, and partial_record.
- Any data from the XACT group is written to table XACT_BAR. Because no table space information is supplied, the table will be created in a table space selected by the system, based on the rules described under the IN *tablespaceName* clause. This table includes all elements contained in the xact group.
- No tables are created for connheader or control; all data for these groups are discarded.

```
CREATE EVENT MONITOR BAR
  FOR STATEMENTS, TRANSACTIONS
  WRITE TO TABLE
  STMT(TABLE MYDEPT.MYSTMTINFO IN MYTABLESPACE
    INCLUDES(ROWS_READ, ROWS_WRITTEN, STMT_TEXT)),
  SUBSECTION(TABLE MYDEPT.MYSUBSECTIONINFO IN MYTABLESPACE
    EXCLUDES(START_TIME, STOP_TIME, PARTIAL_RECORD)),
  XACT
```

Related reference:

- "Basic predicate" in the *SQL Reference, Volume 1*
- "Event monitor logical data groups and data elements" in the *System Monitor Guide and Reference*

CREATE FUNCTION

This statement is used to register or define a user-defined function or function template with an application server.

There are five different types of functions that can be created using this statement. Each of these is described separately.

- External Scalar. The function is written in a programming language and returns a scalar value. The external executable is registered in the database, along with various attributes of the function.
- External Table. The function is written in a programming language and returns a complete table. The external executable is registered in the database along with various attributes of the function.
- OLE DB External Table. A user-defined OLE DB external table function is registered in the database to access data from an OLE DB provider.
- Sourced or Template. A source function is implemented by invoking another function (either built-in, external, SQL, or source) that is already registered in the database.

  It is possible to create a partial function, called a *function template*, which defines what types of values are to be returned, but which contains no executable code. The user maps it to a data source function within a federated system, so that the data source function can be invoked from a federated database. A function template can be registered only with an application server that is designated as a federated server.
- SQL Scalar, Table or Row. The function body is written in SQL and defined together with the registration in the database. It returns a scalar value, a table, or a single row.

Related reference:

- "CREATE FUNCTION (OLE DB External Table)" on page 235
- "CREATE FUNCTION (SQL Scalar, Table or Row)" on page 254
- "CREATE FUNCTION (External Scalar)" on page 190
- "CREATE FUNCTION (External Table)" on page 217
- "CREATE FUNCTION (Sourced or Template)" on page 243

Related samples:

- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"
- "udfcli.sqc -- Call a variety of types of user-defined functions (C)"
- "udfemcli.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"
- "udfcli.c -- How to work with different types of user-defined functions (UDFs) (CLI)"

- “udfcli.sqC -- Call a variety of types of user-defined functions (C++)”
- “udfemcli.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)”
- “UDFCreate.db2 -- How to catalog the Java UDFs contained in UDFsrv.java ”
- “UDFjCreate.db2 -- How to catalog the Java UDFs contained in UDFjsrv.java ”

CREATE FUNCTION (External Scalar)

This statement is used to register a user-defined external scalar function with an application server. A *scalar function* returns a single value each time it is invoked, and is in general valid wherever an SQL expression is valid

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATE_EXTERNAL_ROUTINE authority on the database and at least one of:
  - IMPLICIT_SCHEMA authority on the database, if the schema name of the function does not refer to an existing schema.
  - CREATEIN privilege on the schema, if the schema name of the function refers to an existing schema.

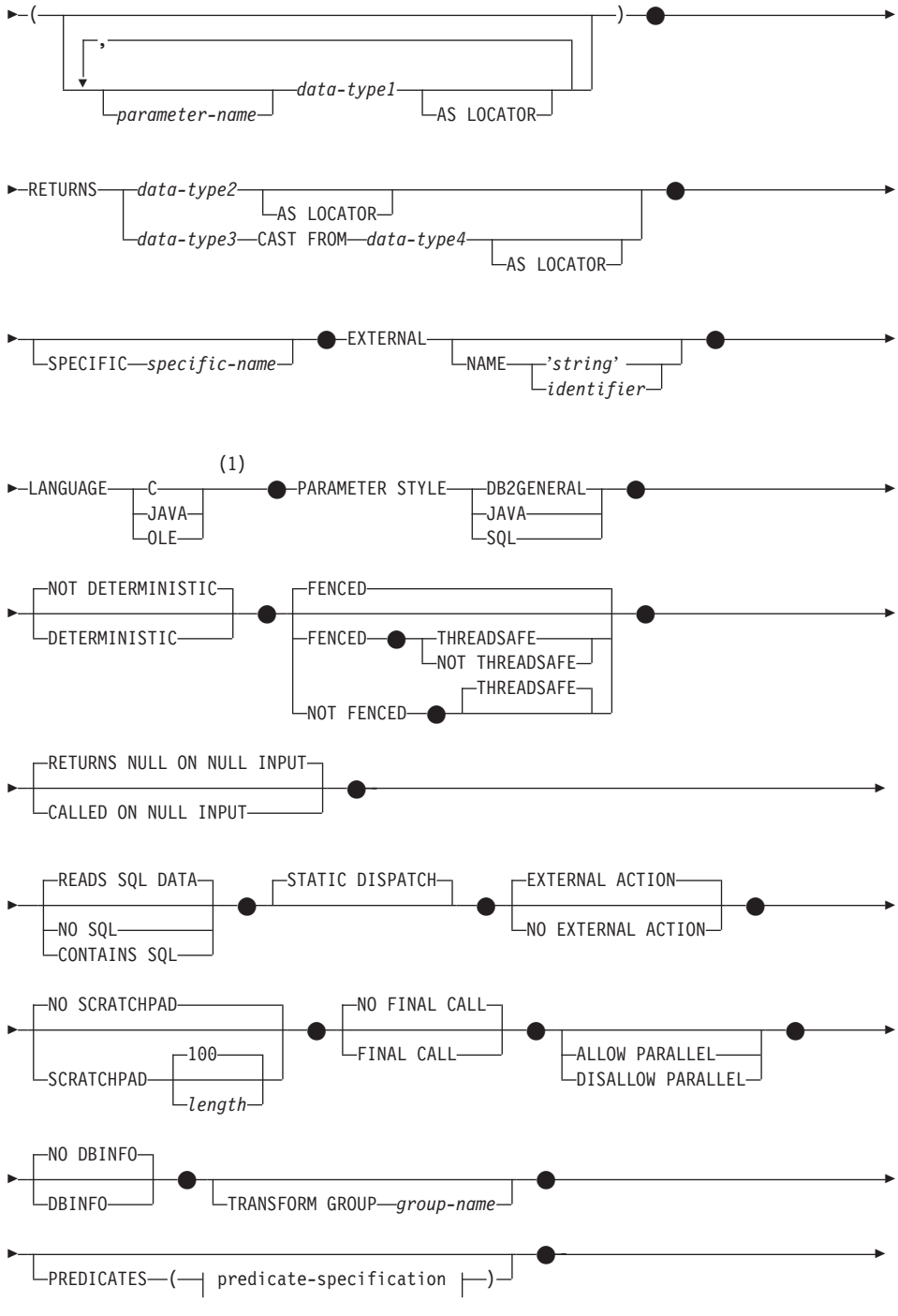
To create a not-fenced function, the privileges held by the authorization ID of the statement must also include at least one of the following:

- CREATE_NOT_FENCED_ROUTINE authority on the database
- SYSADM or DBADM authority.

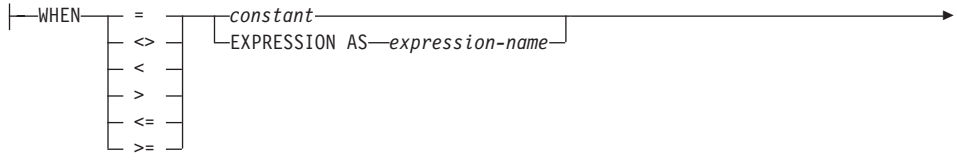
To create a fenced function, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:

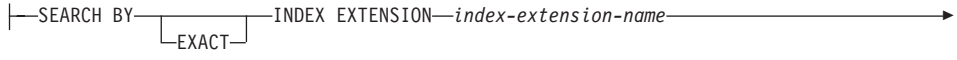

(
,
parameter-name
data-type1
AS LOCATOR
)
RETURNS
data-type2
AS LOCATOR
data-type3
CAST FROM
data-type4
AS LOCATOR
SPECIFIC
specific-name
EXTERNAL
NAME
'string'
identifier
(1)
LANGUAGE
C
JAVA
OLE
PARAMETER STYLE
DB2GENERAL
JAVA
SQL
NOT DETERMINISTIC
DETERMINISTIC
FENCED
FENCED
THREADSAFE
NOT THREADSAFE
NOT FENCED
THREADSAFE
RETURNS NULL ON NULL INPUT
CALLED ON NULL INPUT
READS SQL DATA
NO SQL
CONTAINS SQL
STATIC DISPATCH
EXTERNAL ACTION
NO EXTERNAL ACTION
NO SCRATCHPAD
SCRATCHPAD
100
length
NO FINAL CALL
FINAL CALL
ALLOW PARALLEL
DISALLOW PARALLEL
NO DBINFO
DBINFO
TRANSFORM GROUP
group-name
PREDICATES
(
predicate-specification
)

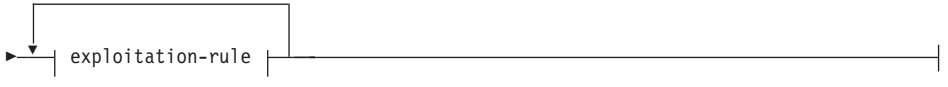
predicate-specification:



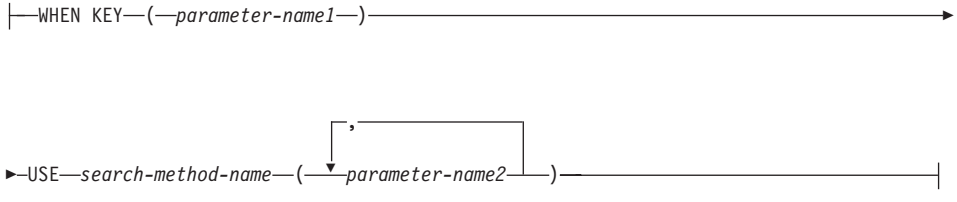
data-filter:

index-exploitation:





exploitation-rule:



Notes:

1 LANGUAGE SQL is also supported.

Description:

function-name
Names the function being defined. It is a qualified or unqualified name

that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier. The qualified name must not be the same as the data type of the first parameter, if that first parameter is a structured type.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function or method described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS';. Otherwise, an error (SQLSTATE 42939) is raised.

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators. Failure to observe this rule will lead to an error (SQLSTATE 42939).

In general, the same name can be used for more than one function if there is some difference in the signature of the functions.

Although there is no prohibition against it, an external user-defined function should not be given the same name as a built-in function, unless it is an intentional override. To give a function having a different meaning the same name (for example, LENGTH, VALUE, MAX), with consistent arguments, as a built-in scalar or column function, is to invite trouble for dynamic SQL statements, or when static SQL applications are rebound; the application may fail, or perhaps worse, may appear to run successfully while providing a different result.

parameter-name
Names the parameter that can be used in the subsequent function definition. Parameter names are required to reference the parameters of a function in the *index-exploitation* clause of a predicate specification.

(data-type1,...)
Identifies the number of input parameters of the function, and specifies the data type of each parameter. One entry in the list must be specified for each parameter that the function will expect to receive. No more than 90 parameters are allowed. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example:

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions, and scales are not considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL (4,3). For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

The second and fourth statements would fail because they are considered to be duplicate functions.

data-type1
Specifies the data type of the parameter.

- SQL data type specifications and abbreviations which may be specified in the *data-type1* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the function may be specified.
- DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815).
- REF(*type-name*) may be specified as the type of a parameter. However, such a parameter must be unscoped.
- Structured types may be specified, provided that appropriate transform functions exist in the associated transform group.

AS LOCATOR
For the LOB types or distinct types which are based on a LOB type, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed to the UDF instead of the actual value. This saves greatly in the number of bytes passed to the UDF, and may save as well in performance, particularly in the case where only a few bytes of the value are actually of interest to the UDF.

Here is an example which illustrates the use of the AS LOCATOR clause in parameter definitions:

```
CREATE FUNCTION foo (CLOB(10M) AS LOCATOR, IMAGE AS LOCATOR)
  ...
```

which assumes that IMAGE is a distinct type based on one of the LOB types.

Note also that for argument promotion purposes, the AS LOCATOR clause has no effect. In the example the types are considered to be CLOB and IMAGE respectively, which would mean that a CHAR or VARCHAR argument could be passed to the function as the first argument. Likewise, the AS LOCATOR has no effect on the function signature, which is used in matching the function (a) when referenced in DML, by a process called "function resolution", and (b) when referenced in a DDL statement such as COMMENT ON or DROP. In fact the clause may or may not be used in COMMENT ON or DROP with no significance.

An error (SQLSTATE 42601) is raised if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the function is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

RETURNS

This mandatory clause identifies the output of the function.

data-type2

Specifies the data type of the output.

In this case, exactly the same considerations apply as for the parameters of external functions described above under *data-type1* for function parameters.

AS LOCATOR

For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the UDF instead of the actual value.

data-type3 **CAST FROM** *data-type4*

Specifies the data type of the output.

This form of the RETURNS clause is used to return a different data type to the invoking statement from the data type that was returned by the function code. For example, in

```
CREATE FUNCTION GET_HIRE_DATE(CHAR(6))
  RETURNS DATE CAST FROM CHAR(10)
  ...
```

the function code returns a CHAR(10) value to the database manager, which, in turn, converts it to a DATE and passes that value to the invoking statement. The *data-type4* must be castable to the *data-type3* parameter. If it is not castable, an error (SQLSTATE 42880) is raised.

Since the length, precision or scale for *data-type3* can be inferred from *data-type4,* it not necessary (but still permitted) to specify the length, precision, or scale for parameterized types specified for *data-type3*. Instead empty parentheses may be used (for example VARCHAR() may be used). FLOAT() cannot be used (SQLSTATE 42601) since parameter value indicates different data types (REAL or DOUBLE).

Distinct types and structured types are not valid as the type specified in *data-type4* (SQLSTATE 42815).

The cast operation is also subject to run-time checks that might result in conversion errors being raised.

AS LOCATOR
For *data-type4* specifications that are LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed back from the UDF instead of the actual value.

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance or method specification that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

EXTERNAL
This clause indicates that the CREATE FUNCTION statement is being

used to register a new function based on code written in an external programming language and adhering to the documented linkage conventions and interface.

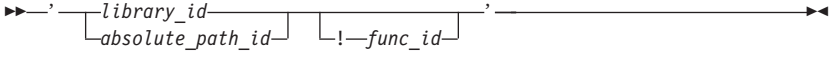
If NAME clause is not specified "NAME *function-name*" is assumed.

NAME *'string'*
This clause identifies the name of the user-written code which implements the function being defined.

The 'string' option is a string constant with a maximum of 254 characters. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

  The *string* specified is the library name and function within library, which the database manager invokes to execute the user-defined function being CREATEd. The library (and the function within the library) do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the library and function within the library must exist and be accessible from the database server machine; otherwise, an error (SQLSTATE 42724) is returned.



  Extraneous blanks are not permitted within the single quotation marks.

  library_id
  Identifies the library name containing the function. The database manager will look for the library in the .../sqllib/function directory (UNIX-based systems), or the ...*instance_name*\function directory (Windows operating systems, as specified by the DB2INSTPROF registry variable), where the database manager will locate the controlling sqllib directory that is being used to run the database manager. For example, the controlling sqllib directory on UNIX-based systems is /u/$DB2INSTANCE/sqllib.

  If 'myfunc' were the *library_id* on a UNIX-based system it would cause the database manager to look for the function in library /u/production/sqllib/function/myfunc, provided the database manager is being run from /u/production.

  On Windows operating systems, the database manager will look in the LIBPATH or PATH if the *library_id* is not located in the function directory.

absolute_path_id
Identifies the full path name of the file containing the function.

On a UNIX-based system, for example, '/u/jchui/mylib/myfunc' would cause the database manager to look in /u/jchui/mylib for the myfunc shared library.

On Windows operating systems, 'd:\mylib\myfunc' would cause the database manager to load dynamic link library, myfunc.dll file, from the d:\mylib directory.

! func_id
Identifies the entry point name of the function to be invoked. The ! serves as a delimiter between the library id and the function id. If *! func_id* is omitted, the database manager will use the default entry point established when the library was linked.

In a UNIX-based system, for example, 'mymod!func8' would direct the database manager to look for the library $inst_home_dir/sqllib/function/mymod and to use entry point func8 within that library.

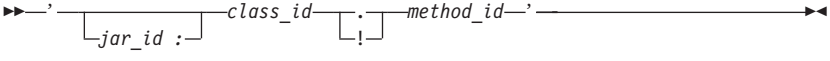
On Windows operating systems, 'mymod!func8' would direct the database manager to load the mymod.dll file and call the func8() function in the dynamic link library (DLL).

If the string is not properly formed, an error (SQLSTATE 42878) is raised.

The body of every external function should be in a directory that is available on every partition of the database.

- For LANGUAGE JAVA:

  The *string* specified contains the optional jar file identifier, class identifier and method identifier, which the database manager invokes to execute the user-defined function being CREATEd. The class identifier and method identifier do not need to exist when the CREATE FUNCTION statement is performed. If a *jar_id* is specified, it must exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42724) is raised.



Extraneous blanks are not permitted within the single quotes.

jar_id
Identifies the jar identifier given to the jar collection when it was installed in the database. It can be either a simple identifier, or a schema qualified identifier. Examples are 'myJar' and 'mySchema.myJar'.

class_id
Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix, for example, 'myPacks.UserFuncs'. The Java virtual machine will look in directory '.../myPacks/UserFuncs/' for the classes. On Windows operating systems, the Java virtual machine will look in directory '...\myPacks\UserFuncs\'.

method_id
Identifies the method name of the Java object to be invoked.

- For LANGUAGE OLE:

  The *string* specified is the OLE programmatic identifier (progid) or class identifier (clsid), and method identifier, which the database manager invokes to execute the user-defined function being CREATEd. The programmatic identifier or class identifier, and method identifier do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42724) is raised.

  Extraneous blanks are not permitted within the single quotes.

  progid
  Identifies the programmatic identifier of the OLE object.

  progid is not interpreted by the database manager but only forwarded to the OLE APIs at run time. The specified OLE object must be creatable and support late binding (also called IDispatch-based binding).

  clsid
  Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a *progid* in the case that an OLE object is not registered with a progid. The *clsid* has the form:

  {nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}

where 'n' is an alphanumeric character. *clsid* is not interpreted by the database manager but only forwarded to the OLE APIs at run time.

method_id
Identifies the method name of the OLE object to be invoked.

NAME *identifier*
This *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the user-defined function body is written.

C This means the database manager will call the user-defined function as if it were a C function. The user-defined function must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA This means the database manager will call the user-defined function as a method in a Java class.

OLE This means the database manager will call the user-defined function as if it were a method exposed by an OLE automation object. The user-defined function must conform with the OLE automation data types and invocation mechanism, as described in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is only supported for user-defined functions stored in DB2 for Windows operating systems. THREADSAFE may not be specified for UDFs defined with LANGUAGE OLE (SQLSTATE 42613).

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

DB2GENERAL
Used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a Java class. This can only specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

JAVA
This means that the function will use a parameter passing convention

that conforms to the Java language and SQLj Routines specification. This can only be specified when LANGUAGE JAVA is used and there are no structured types as parameter or return types (SQLSTATE 429B8). PARAMETER STYLE JAVA functions do not support the FINAL CALL, SCRATCHPAD or DBINFO clauses.

SQL
Used to specify the conventions for passing parameters to and returning the value from external functions that conform to C language calling and linkage conventions or methods exposed by OLE automation objects. This must be specified when LANGUAGE C or LANGUAGE OLE is used.

DETERMINISTIC or **NOT DETERMINISTIC**
This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a NOT DETERMINISTIC function would be a random-number generator. An example of a DETERMINISTIC function would be a function that determines the square root of the input.

FENCED or **NOT FENCED**
This clause specifies whether or not the function is considered "safe" to run in the database manager operating environment's process or address space.

If a function is registered as FENCED, the database manager protects its internal resources (for example, data buffers) from access by the function. Most functions will have the option of running as FENCED or NOT FENCED. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION:
Use of NOT FENCED for functions not adequately coded, reviewed and tested can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user-defined functions are used.

Only FENCED can be specified for a function with LANGUAGE OLE or NOT THREADSAFE (SQLSTATE 42613).

If the function is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

Either SYSADM authority, DBADM authority, or a special authority (CREATE_NOT_FENCED_ROUTINE) is required to register a user-defined function as NOT FENCED.

THREADSAFE or **NOT THREADSAFE**
Specifies whether the function is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the function is defined with LANGUAGE other than OLE:

- If the function is defined as THREADSAFE, the database manager can invoke the function in the same process as other routines. In general, to be threadsafe, a function should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED functions can be THREADSAFE.
- If the function is defined as NOT THREADSAFE, the database manager will never invoke the function in the same process as another routine.

For FENCED functions, THREADSAFE is the default if the LANGUAGE is JAVA. For all other languages, NOT THREADSAFE is the default. If the function is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).

For NOT FENCED functions, THREADSAFE is the default. NOT THREADSAFE cannot be specified (SQLSTATE 42613).

RETURNS NULL ON NULL INPUT or **CALLED ON NULL INPUT**
This optional clause may be used to avoid a call to the external function if any of the arguments is null. If the user-defined function is defined to have no parameters, then of course this null argument condition cannot arise, and it does not matter how this specification is coded.

If RETURNS NULL ON NULL INPUT is specified, and if, at execution time, any one of the function's arguments is null, then the user-defined function is not called and the result is the null value.

If CALLED ON NULL INPUT is specified, then regardless of whether any arguments are null, the user-defined function is called. It can return a null value or a normal (non-null) value. But responsibility for testing for null argument values lies with the UDF.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

NO SQL, CONTAINS SQL, READS SQL DATA
Indicates whether the function issues any SQL statements and, if so, what type.

NO SQL
Indicates that the function cannot execute any SQL statements (SQLSTATE 38001).

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the function (SQLSTATE 38004 or 42985). Statements that are not supported in any function return a different error (SQLSTATE 38003 or 42985).

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the function (SQLSTATE 38002 or 42985). Statements that are not supported in any function return a different error (SQLSTATE 38003 or 42985).

STATIC DISPATCH
This optional clause indicates that at function resolution time, DB2 chooses a function based on the static types (declared types) of the parameters of the function.

NO EXTERNAL ACTION or **EXTERNAL ACTION**
This optional clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. Optimizations that assume functions have no external impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

NO SCRATCHPAD or **SCRATCHPAD** *length*
This optional clause may be used to specify whether a scratchpad is to be provided for an external function. (It is strongly recommended that user-defined functions be re-entrant, so a scratchpad provides a means for the function to "save state" from one call to the next.)

If SCRATCHPAD is specified, then at first invocation of the user-defined function, memory is allocated for a scratchpad to be used by the external function. This scratchpad has the following characteristics:

- *length*, if specified, sets the size of the scratchpad in bytes; this value must be between 1 and 32 767 (SQLSTATE 42820). The default size is 100 bytes.
- It is initialized to all X'00''s.
- Its scope is the SQL statement. There is one scratchpad per reference to the external function in the SQL statement. So if the UDFX function in the following statement is defined with the SCRATCHPAD keyword, three scratchpads would be assigned.

```
SELECT A, UDFX(A) FROM TABLEB
  WHERE UDFX(A) > 103 OR UDFX(A) < 19
```

If ALLOW PARALLEL is specified or defaulted to, then the scope is different from the above. If the function is executed in multiple partitions, a scratchpad would be assigned in each partition where the function is processed, for each reference to the function in the SQL statement. Similarly, if the query is executed with intra-partition parallelism enabled, more than three scratchpads may be assigned.

- It is persistent. Its content is preserved from one external function call to the next. Any changes made to the scratchpad by the external function on one call will be there on the next call. The database manager initializes scratchpads at the beginning of execution of each SQL statement. The database manager may reset scratchpads at the beginning of execution of each subquery. The system issues a final call before resetting a scratchpad if the FINAL CALL option is specified.
- It can be used as a central point for system resources (for example, memory) which the external function might acquire. The function could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

  (In such a case where system resource is acquired, the FINAL CALL keyword should also be specified; this causes a special call to be made at end-of-statement to allow the external function to free any system resources acquired.)

If SCRATCHPAD is specified, then on each invocation of the user-defined function an additional argument is passed to the external function which addresses the scratchpad.

If NO SCRATCHPAD is specified then no scratchpad is allocated or passed to the external function.

SCRATCHPAD is not supported for PARAMETER STYLE JAVA functions.

FINAL CALL or **NO FINAL CALL**

This optional clause specifies whether a final call is to be made to an external function. The purpose of such a final call is to enable the external function to free any system resources it has acquired. It can be useful in conjunction with the SCRATCHPAD keyword in situations where the external function acquires system resources such as memory and anchors them in the scratchpad. If FINAL CALL is specified, then at execution time:

- An additional argument is passed to the external function which specifies the type of call. The types of calls are:
  - Normal call: SQL arguments are passed and a result is expected to be returned.

– First call: the first call to the external function for this reference to the user-defined function in this SQL statement. The first call is a normal call.
– Final call: a final call to the external function to enable the function to free up resources. The final call is not a normal call. This final call occurs at the following times:
  - End-of-statement: This case occurs when the cursor is closed for cursor-oriented statements, or when the statement is through executing otherwise.
  - End-of-parallel-task: This case occurs when the function is executed by parallel tasks.
  - End-of-transaction or interrupt: This case occurs when the normal end-of-statement does not occur. For example, the logic of an application may for some reason bypass the close of the cursor. During this type of final call, no SQL statements may be issued except for CLOSE cursor (SQLSTATE 38505). This type of final call is indicated with a special value in the ″call type″ argument.

  If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made at the subsequent close of the cursor or at the end of the application.

If NO FINAL CALL is specified then no “call type” argument is passed to the external function, and no final call is made.

FINAL CALL is not supported for PARAMETER STYLE JAVA functions.

ALLOW PARALLEL or **DISALLOW PARALLEL**
This optional clause specifies whether, for a single reference to the function, the invocation of the function can be parallelized. In general, the invocations of most scalar functions should be parallelizable, but there may be functions (such as those depending on a single copy of a scratchpad) that cannot. If either ALLOW PARALLEL or DISALLOW PARALLEL are specified for a scalar function, then DB2 will accept this specification. The following questions should be considered in determining which keyword is appropriate for the function.

- Are all the UDF invocations completely independent of each other? If YES, then specify ALLOW PARALLEL.
- Does each UDF invocation update the scratchpad, providing value(s) that are of interest to the next invocation? (For example, the incrementing of a counter.) If YES, then specify DISALLOW PARALLEL or accept the default.
- Is there some external action performed by the UDF which should happen only on one partition? If YES, then specify DISALLOW PARALLEL or accept the default.

- Is the scratchpad used, but only so that some expensive initialization processing can be performed a minimal number of times? If YES, then specify ALLOW PARALLEL.

In any case, the body of every external function should be in a directory that is available on every partition of the database.

The default value is ALLOW PARALLEL, except if one or more of the following options is specified in the statement.

- NOT DETERMINISTIC
- EXTERNAL ACTION
- SCRATCHPAD
- FINAL CALL

If any of these options is specified or implied, the default value is DISALLOW PARALLEL.

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the function will inherit their initial values from the environment of the invoking statement. For a function invoked in the select-statement of a cursor, the initial values are inherited from the environment when the cursor is opened. For a routine invoked in a nested object (for example a trigger or view), the initial values are inherited from the runtime environment (not inherited from the object definition).

No changes to the special registers are passed back to the invoker of the function.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

NOT FEDERATED
This optional clause indicates that federated objects cannot be used in any SQL statement in the function. Using a federated object will result in an error (SQLSTATE 55047).

NO DBINFO or **DBINFO**
This optional clause specifies whether certain specific information known by DB2 will be passed to the UDF as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613) or PARAMETER STYLE JAVA.

If DBINFO is specified, then a structure is passed to the UDF which contains the following information:

- Data base name - the name of the currently connected database.

- Application ID - unique application ID which is established for each connection to the database.
- Application Authorization ID - the application run-time authorization ID, regardless of the nested UDFs in between this UDF and the application.
- Code page - identifies the database code page.
- Schema name - under the exact same conditions as for Table name, contains the name of the schema; otherwise blank.
- Table name - if and only if the UDF reference is either the right-hand side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement, contains the unqualified name of the table being updated or inserted; otherwise blank.
- Column name - under the exact same conditions as for Table name, contains the name of the column being updated or inserted; otherwise blank.
- Database version/release - identifies the version, release and modification level of the database server invoking the UDF.
- Platform - contains the server's platform type.
- Table function result column numbers - not applicable to external scalar functions.

TRANSFORM GROUP *group-name*

Indicates the transform group to be used for user-defined structured type transformations when invoking the function. A transform is required if the function definition includes a user-defined structured type as either a parameter or returns data type. If this clause is not specified, the default group name DB2_FUNCTION is used. If the specified (or default) *group-name* is not defined for a referenced structured type, an error is raised (SQLSTATE 42741). If a required FROM SQL or TO SQL transform function is not defined for the given group-name and structured type, an error is raised (SQLSTATE 42744).

The transform functions, both FROM SQL and TO SQL, whether designated or implied, must be SQL functions which properly transform between the structured type and its built in type attributes.

PREDICATES

Defines the filtering and/or index extension exploitation performed when this function is used in a predicate. A predicate-specification allows the optional SELECTIVITY clause of a search-condition to be specified. If the PREDICATES clause is specified, the function must be defined as DETERMINISTIC with NO EXTERNAL ACTION (SQLSTATE 42613).

WHEN *comparison-operator*

Introduces a specific use of the function in a predicate with a comparison operator ("=", "<", ">", ">=", "<=", "<>").

constant
Specifies a constant value with a data type comparable to the RETURNS type of the function (SQLSTATE 42818). When a predicate uses this function with the same comparison operator and this constant, the specified filtering and index exploitation will be considered by the optimizer.

EXPRESSION AS *expression-name*
Provides a name for an expression. When a predicate uses this function with the same comparison operator and an expression, filtering and index exploitation may be used. The expression is assigned an expression name so that it can be used as a search function argument. The *expression-name* cannot be the same as any *parameter-name* of the function being created (SQLSTATE 42711). When an expression is specified, the type of the expression is identified.

FILTER USING
Allows specification of an external function or a case expression to be used for additional filtering of the result table.

function-invocation
Specifies a filter function that can be used to perform additional filtering of the result table. This is a version of the defined function (used in the predicate) that reduces the number of rows on which the user-defined predicate must be executed, to determine if rows qualify. If the results produced by the index are close to the results expected for the user-defined predicate, applying the filtering function may be redundant. If not specified, data filtering is not performed.

This function can use any *parameter-name*, the *expression-name*, or constants as arguments (SQLSTATE 42703), and returns an integer (SQLSTATE 428E4). A return value of 1 means the row is kept, otherwise it is discarded.

This function must also:

- not be defined with LANGUAGE SQL (SQLSTATE 429B4)
- not be defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 42845)
- not have a structured data type as the data type of any of the parameters (SQLSTATE 428E3)
- not include a subquery (SQLSTATE 428E4).

If an argument invokes another function or method, these four rules are also enforced for this nested function or method. However, system generated observer methods are allowed as

arguments to the filter function (or any function or method used as an argument), as long as the argument evaluates to a built-in data type.

The definer of the function must have EXECUTE privilege on the specified filter function.

case-expression
Specifies a case expression for additional filtering of the result table. The *searched-when-clause* and *simple-when-clause* can use *parameter-name*, *expression-name*, or a constant (SQLSTATE 42703). An external function with the rules specified in FILTER USING *function-invocation* may be used as a result-expression. Any function or method referenced in the case-expression must also conform to the four rules listed under *function-invocation*.

Subqueries cannot be used anywhere in the *case-expression* (SQLSTATE 428E4).

The case expression must return an integer (SQLSTATE 428E4). A return value of 1 in the result-expression means that the row is kept, otherwise it is discarded.

index-exploitation
Defines a set of rules in terms of the search method of an index extension that can be used to exploit the index.

SEARCH BY INDEX EXTENSION *index-extension-name*
Identifies the index extension. The *index-extension-name* must identify an existing index extension.

EXACT
Indicates that the index lookup is exact in terms of the predicate evaluation. Use EXACT to tell DB2 that neither the original user-defined predicate function or the filter need to be applied after the index lookup. The EXACT predicate is useful when the index lookup returns the same results as the predicate.

If EXACT is not specified, then the original user-defined predicate is applied after index lookup. If the index is expected to provide only an approximation of the predicate, do not specify the EXACT option.

If the index lookup is not used, then the filter function and the original predicate have to be applied.

exploitation-rule
Describes the search targets and search arguments and how they can be used to perform the index search through a search method defined in the index extension.

WHEN KEY (*parameter-name1***)**
This defines the search target. Only one search target can be specified for a key. The *parameter-name1* value identifies parameter names of the defined function (SQLSTATE 42703 or 428E8).

The data type of *parameter-name1* must match that of the source key specified in the index extension (SQLSTATE 428EY). The match must be exact for built-in and distinct data types and within the same structured type hierarchy for structured types.

This clause is true when the values of the named parameter are columns that are covered by an index based on the index extension specified.

USE *search-method-name***(***parameter-name2,...***)**
This defines the search argument. It identifies which search method to use from those defined in the index extension. The *search-method-name* must match a search method defined in the index extension (SQLSTATE 42743). The *parameter-name2* values identify parameter names of the defined function or the *expression-name* in the EXPRESSION AS clause (SQLSTATE 42703). It must be different from any parameter name specified in the search target (SQLSTATE 428E9). The number of parameters and the data type of each *parameter-name2* must match the parameters defined for the search method in the index extension (SQLSTATE 42816). The match must be exact for built-in and distinct data types and within the same structured type hierarchy for structured types.

Notes:

- ***Compatibilities***
  - For compatibility with DB2 UDB for OS/390 and z/OS:
    - The following syntax is accepted as the default behavior:
      - ASUTIME NO LIMIT
      - NO COLLID
      - PROGRAM TYPE SUB
      - STAY RESIDENT NO
  - For compatibility with previous versions of DB2:
    - PARAMETER STYLE DB2SQL can be specified in place of PARAMETER STYLE SQL
    - NOT VARIANT can be specified in place of DETERMINISTIC, and VARIANT can be specified in place of NOT DETERMINISTIC
    - NULL CALL can be specified in place of CALLED ON NULL INPUT, and NOT NULL CALL can be specified in place of RETURNS NULL ON NULL INPUT

- Determining whether one data type is castable to another data type does not consider length or precision and scale for parameterized data types such as CHAR and DECIMAL. Therefore, errors may occur when using a function as a result of attempting to cast a value of the source data type to a value of the target data type. For example, VARCHAR is castable to DATE but if the source type is actually defined as VARCHAR(5), an error will occur when using the function.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values (see "Promotion of data types"). For example, a constant which may be used as an input value could have a built-in data type different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:
  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- For portability of UDFs across platforms the following data types should not be used:
  - FLOAT- use DOUBLE or REAL instead.
  - NUMERIC- use DECIMAL instead.
  - LONG VARCHAR- use CLOB (or BLOB) instead.
- A function and a method may not be in an overriding relationship (SQLSTATE 42745). For more information about overriding, see "CREATE TYPE (Structured)".
- A function may not have the same signature as a method (comparing the first *parameter-type* of the function with the *subject-type* of the method) (SQLSTATE 42723).
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- Only routines defined as NO SQL can be used to define an index extension (SQLSTATE 428F8).
- ***Table access restrictions***

  If a function is defined as READS SQL DATA, no statement in the function can access a table that is being modified by the statement which invoked the function (SQLSTATE 57053). For example, suppose the user-defined function BONUS() is defined as READS SQL DATA. If the statement UPDATE EMPLOYEE SET SALARY = SALARY + BONUS(EMPNO) is invoked, no SQL statement in the BONUS function can read from the EMPLOYEE table.
- ***Privileges***

- The definer of a function always receives the EXECUTE privilege WITH GRANT OPTION on the function, as well as the right to drop the function.
- When the function is used in an SQL statement, the function definer must have the EXECUTE privilege on any packages used by the function.

Examples:

Example 1: Pellow is registering the CENTRE function in his PELLOW schema. Let those keywords that will default do so, and let the system provide a function specific name:

```
CREATE FUNCTION CENTRE (INT,FLOAT)
  RETURNS FLOAT
  EXTERNAL NAME 'mod!middle'
  LANGUAGE C
  PARAMETER STYLE SQL
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
```

Example 2: Now, McBride (who has DBADM authority) is registering another CENTRE function in the PELLOW schema, giving it an explicit specific name for subsequent data definition language use, and explicitly providing all keyword values. Note also that this function uses a scratchpad and presumably is accumulating data there that affects subsequent results. Since DISALLOW PARALLEL is specified, any reference to the function is not parallelized and therefore a single scratchpad is used to perform some one-time only initialization and save the results.

```
CREATE FUNCTION PELLOW.CENTRE (FLOAT, FLOAT, FLOAT)
  RETURNS DECIMAL(8,4) CAST FROM FLOAT
  SPECIFIC FOCUS92
  EXTERNAL NAME 'effects!focalpt'
  LANGUAGE C   PARAMETER STYLE SQL
  DETERMINISTIC   FENCED   NOT NULL CALL   NO SQL   NO EXTERNAL ACTION
  SCRATCHPAD   NO FINAL CALL
  DISALLOW PARALLEL
```

Example 3: The following is the C language user-defined function program written to implement the rule:

```
output = 2 * input - 4
```

returning NULL if and only if the input is null. It could be written even more simply (that is, without null checking), if the CREATE FUNCTION statement had used NOT NULL CALL. The CREATE FUNCTION statement:

```
CREATE FUNCTION ntest1 (SMALLINT)
  RETURNS SMALLINT
  EXTERNAL NAME 'ntest1!nudft1'
  LANGUAGE C    PARAMETER STYLE SQL
  DETERMINISTIC   NOT FENCED   NULL CALL
  NO SQL   NO EXTERNAL ACTION
```

The program code:

```
#include "sqlsystm.h"
/* NUDFT1 IS A USER_DEFINED SCALAR FUNCTION */
/* udft1 accepts smallint input
and produces smallint output
implementing the rule:
if (input is null)
set output = null;
else
set output = 2 * input - 4;
*/
void SQL_API_FN nudft1
(short *input,      /* ptr to input arg */
short *output,     /* ptr to where result goes */
short *input_ind,  /* ptr to input indicator var */
short *output_ind, /* ptr to output indicator var */
char sqlstate[6],  /* sqlstate, allows for null-term */
char fname[28],    /* fully qual func name, nul-term */
char finst[19],    /* func specific name,  null-term */
char msgtext[71])  /* msg text buffer,     null-term */
{
/* first test for null input */
if (*input_ind == -1)
{
/* input is null, likewise output */
*output_ind = -1;
}
else
{
/* input is not null.  set output to 2*input-4 */
*output = 2 * (*input) - 4;
/* and set out null indicator to zero */
*output_ind = 0;
}
/* signal successful completion by leaving sqlstate as is */
/* and exit */
return;
}
/* end of UDF: NUDFT1 */
```

Example 4: The following registers a Java UDF which returns the position of the first vowel in a string. The UDF is written in Java, is to be run fenced, and is the findvwl method of class javaUDFs.

```
CREATE FUNCTION findv ( CLOB(100K))
  RETURNS INTEGER
  FENCED
```

```
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'javaUDFs.findvwl'
NO EXTERNAL ACTION
CALLED ON NULL INPUT
DETERMINISTIC
NO SQL
```

Example 5: This example outlines a user-defined predicate WITHIN that takes two parameters, g1 and g2, of type SHAPE as input:

```
CREATE FUNCTION within (g1 SHAPE, g2 SHAPE)
RETURNS INTEGER
LANGUAGE C
PARAMETER STYLE SQL
NOT VARIANT
NOT FENCED
NO SQL
NO EXTERNAL ACTION
EXTERNAL NAME 'db2sefn!SDESpatilRelations'
PREDICATES
WHEN = 1
FILTER USING mbrOverlap(g1..xmin, g1..ymin, g1..xmax, g1..max,
g2..xmin, g2..ymin, g2..xmax, g2..ymax)
SEARCH BY INDEX EXTENSION gridIndex
WHEN KEY(g1) USE withinExplRule(g2)
WHEN KEY(g2) USE withinExplRule(g1)
```

The description of the WITHIN function is similar to that of any user-defined function, but the following additions indicate that this function can be used in a user-defined predicate.

- **PREDICATES WHEN = 1** indicates that when this function appears as

  ```
  within(g1, g2) = 1
  ```

  in the WHERE clause of a DML statement, the predicate is to be treated as a user-defined predicate and the index defined by the index extension *gridIndex* should be used to retrieve rows that satisfy this predicate. If a constant is specified, the constant specified during the DML statement has to match exactly the constant specified in the create index statement. This condition is provided mainly to cover Boolean expression where the result type is either a 1 or a 0. For other cases, the EXPRESSION clause is a better choice.
- **FILTER USING mbrOverlap** refers to a filtering function mbrOverlap, which is a cheaper version of the WITHIN predicate. In the above example, the mbrOverlap function takes the minimum bounding rectangles as input and quickly determines if they overlap or not. If the minimum bounding rectangles of the two input shapes do not overlap, then g1 will not be contained with g2. Therefore the tuple can be safely discarded, avoiding the application of the expensive WITHIN predicate.

- The **SEARCH BY INDEX EXTENSION** clause indicates that combinations of index extension and search target can be used for this user-defined predicate.

Example 6: This example outlines a user-defined predicate DISTANCE that takes two parameters, P1 and P2, of type POINT as input:

```
CREATE FUNCTION distance (P1 POINT, P2 POINT)
  RETURNS INTEGER
  LANGUAGE C
  PARAMETER STYLE SQL
  NOT VARIANT
  NOT FENCED
  NO SQL
  NO EXTERNAL ACTION
  EXTERNAL NAME 'db2sefn!SDEDistances'
  PREDICATES
  WHEN > EXPRESSION AS distExpr
  SEARCH BY INDEX EXTENSION gridIndex
  WHEN KEY(P1) USE distanceGrRule(P2, distExpr)
  WHEN KEY(P2) USE distanceGrRule(P1, distExpr)
```

The description of the DISTANCE function is similar to that of any user-defined function, but the following additions indicate that when this function is used in a predicate, that predicate is a user-defined predicate.

- **PREDICATES WHEN > EXPRESSION AS distExpr** is another valid predicate specification. When an expression is specified in the WHEN clause, the result type of that expression is used for determining if the predicate is a user-defined predicate in the DML statement. For example:

  ```
  SELECT T1.C1
    FROM T1, T2
    WHERE distance (T1.P1, T2.P1) > T2.C2
  ```

  The predicate specification distance takes two parameters as input and compares the results with T2.C2, which is of type INTEGER. Since only the data type of the right hand side expression matters, (as opposed to using a specific constant), it is better to choose the EXPRESSION clause in the CREATE FUNCTION DDL for specifying a wildcard as the comparison value.

  Alternatively, the following is also a valid user-defined predicate:

  ```
  SELECT T1.C1
    FROM T1, T2
    WHERE distance(T1.P1, T2.P1) > distance (T1.P2, T2.P2)
  ```

  There is currently a restriction that only the right hand side is treated as the expression; the term on the left hand side is the user-defined function for the user-defined predicate.

- The **SEARCH BY INDEX EXTENSION** clause indicates that combinations of index extension and search target can be used for this user-defined-predicate. In the case of the distance function, the expression identified as distExpr is also one of the search arguments that is passed to the range-producer function (defined as part of the index extension). The expression identifier is used to define a name for the expression so that it is passed to the range-producer function as an argument.

Related reference:

- "Basic predicate" in the *SQL Reference, Volume 1*
- "CREATE TYPE (Structured)" on page 427
- "CREATE FUNCTION (SQL Scalar, Table or Row)" on page 254
- "SQL statements allowed in routines" in the *SQL Reference, Volume 1*
- "Special registers" in the *SQL Reference, Volume 1*
- "Promotion of data types" in the *SQL Reference, Volume 1*
- "Casting between data types" in the *SQL Reference, Volume 1*

CREATE FUNCTION (External Table)

This statement is used to register a user-defined external table function with an application server.

A *table function* may be used in the FROM clause of a SELECT, and returns a table to the SELECT by returning one row at a time.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATE_EXTERNAL_ROUTINE authority on the database and at least one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
  - CREATEIN privilege on the schema, if the schema name of the function exists.

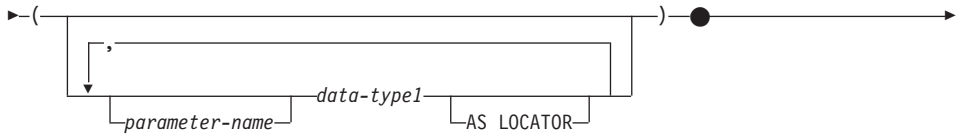
To create a not-fenced function, the privileges held by the authorization ID of the statement must also include at least one of the following:

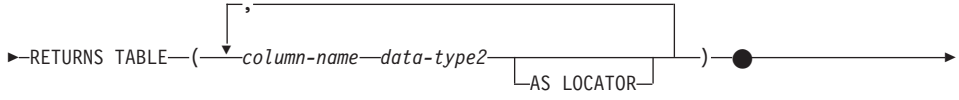
- CREATE_NOT_FENCED_ROUTINE authority on the database
- SYSADM or DBADM authority.

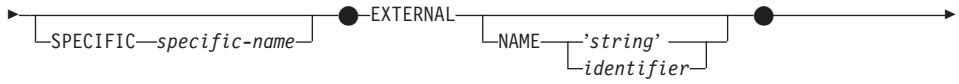
To create a fenced function, no additional authorities or privileges are required.

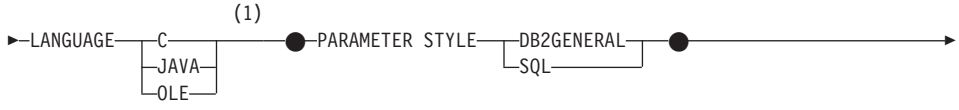
If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

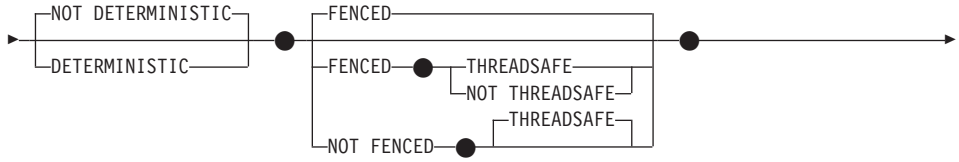
Syntax:

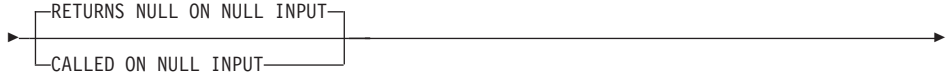

(
,
parameter-name
data-type1
AS LOCATOR
)


RETURNS TABLE
(
,
column-name
data-type2
AS LOCATOR
)

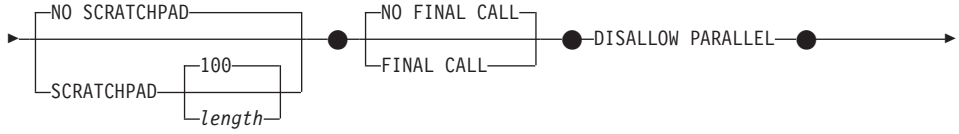

SPECIFIC
specific-name
EXTERNAL
NAME
'string'
identifier

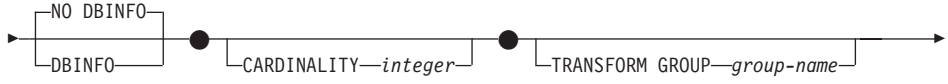

LANGUAGE
C
JAVA
OLE
(1)
PARAMETER STYLE
DB2GENERAL
SQL

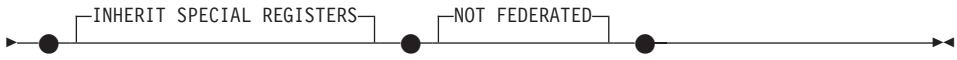

NOT DETERMINISTIC
DETERMINISTIC
FENCED
FENCED
THREADSAFE
NOT THREADSAFE
NOT FENCED
THREADSAFE


RETURNS NULL ON NULL INPUT
CALLED ON NULL INPUT


READS SQL DATA
NO SQL
CONTAINS SQL
STATIC DISPATCH
EXTERNAL ACTION
NO EXTERNAL ACTION


NO SCRATCHPAD
SCRATCHPAD
100
length
NO FINAL CALL
FINAL CALL
DISALLOW PARALLEL


NO DBINFO
DBINFO
CARDINALITY
integer
TRANSFORM GROUP
group-name



Notes:

1 For information on creating LANGUAGE OLE DB external table functions, see "CREATE FUNCTION (OLE DB External Table)". For information on creating LANGUAGE SQL table functions, see "CREATE FUNCTION (SQL Scalar, Table or Row)".

Description:

function-name

Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier. The qualified name must not be the same as the data type of the first parameter, if that first parameter is a structured type.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

The same name can be used for more than one function if there is some difference in the signature of the functions. Although there is no prohibition against it, an external user-defined table function should not be given the same name as a built-in function.

parameter-name

Specifies an optional name for the parameter that is distinct from the names of all other parameters in this function.

(data-type1,...)
Identifies the number of input parameters of the function, and specifies the data type of each parameter. One entry in the list must be specified for each parameter that the function will expect to receive. No more than 90 parameters are allowed. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example,

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions, and scales are not considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL (4,3). For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...

CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

the second and fourth statements would fail because they are considered to be a duplicate functions.

data-type1
Specifies the data type of the parameter.

- SQL data type specifications and abbreviations which may be specified in the *data-type* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the function may be specified.
- DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815).
- REF(*type-name*) may be specified as the data type of a parameter. However, such a parameter must be unscoped (SQLSTATE 42997).
- Structured types may be specified, provided that appropriate transform functions exist in the associated transform group.

AS LOCATOR
For the LOB types or distinct types which are based on a LOB

type, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed to the UDF instead of the actual value. This saves greatly in the number of bytes passed to the UDF, and may save as well in performance, particularly in the case where only a few bytes of the value are actually of interest to the UDF.

Here is an example which illustrates the use of the AS LOCATOR clause in parameter definitions:

```
CREATE FUNCTION foo ( CLOB(10M) AS LOCATOR, IMAGE AS LOCATOR)
  ...
```

which assumes that IMAGE is a distinct type based on one of the LOB types.

Note also that for argument promotion purposes, the AS LOCATOR clause has no effect. In the example the types are considered to be CLOB and IMAGE respectively, which would mean that a CHAR or VARCHAR argument could be passed to the function as the first argument. Likewise, the AS LOCATOR has no effect on the function signature, which is used in matching the function (a) when referenced in DML, by a process called ″function resolution″, and (b) when referenced in a DDL statement such as COMMENT ON or DROP. In fact the clause may or may not be used in COMMENT ON or DROP with no significance.

An error (SQLSTATE 42601) is raised if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the function is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

RETURNS TABLE

Specifies that the output of the function is a table. The parentheses that follow this keyword delimit a list of the names and types of the columns of the table, resembling the style of a simple CREATE TABLE statement which has no additional specifications (constraints, for example). No more than 255 columns are allowed (SQLSTATE 54011).

column-name

Specifies the name of this column. The name cannot be qualified and the same name cannot be used for more than one column of the table.

data-type2

Specifies the data type of the column, and can be any data type supported for a parameter of a UDF written in the particular language, except for structured types (SQLSTATE 42997).

AS LOCATOR
When *data-type2* is a LOB type or distinct type based on a LOB type, the use of this option indicates that the function is returning a locator for the LOB value that is instantiated in the result table.

The valid types for use with this clause are discussed in "CREATE FUNCTION (External Scalar)".

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

EXTERNAL
This clause indicates that the CREATE FUNCTION statement is being used to register a new function based on code written in an external programming language and adhering to the documented linkage conventions and interface.

If NAME clause is not specified "NAME *function-name*" is assumed.

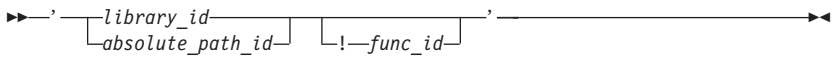
NAME *'string'*
This clause identifies the user-written code which implements the function being defined.

The 'string' option is a string constant with a maximum of 254 characters. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

  The *string* specified is the library name and function within library, which the database manager invokes to execute the user-defined function being CREATEd. The library (and the function within the library) do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an

SQL statement, the library and function within the library must exist and be accessible from the database server machine.



Extraneous blanks are not permitted within the single quotes.

library_id
Identifies the library name containing the function. The database manager will look for the library in the .../sqllib/function directory (UNIX-based systems), or ...*instance_name*\function directory (Windows 32-bit operating systems as specified by the DB2INSTPROF registry variable), where the database manager will locate the controlling sqllib directory which is being used to run the database manager. For example, the controlling sqllib directory in UNIX-based systems is /u/$DB2INSTANCE/sqllib.

If 'myfunc' were the *library_id* in a UNIX-based system it would cause the database manager to look for the function in library /u/production/sqllib/function/myfunc, provided the database manager is being run from /u/production.

For Windows 32-bit operating systems, the database manager will look in the LIBPATH or PATH if the *library_id* is not located in the function directory.

absolute_path_id
Identifies the full path name of the function.

In a UNIX-based system, for example, '/u/jchui/mylib/myfunc' would cause the database manager to look in /u/jchui/mylib for the myfunc function.

On Windows 32-bit operating systems 'd:\mylib\myfunc' would cause the database manager to load the myfunc.dll file from the d:\mylib directory.

! func_id
Identifies the entry point name of the function to be invoked. The ! serves as a delimiter between the library id and the function id. If *! func_id* is omitted, the database manager will use the default entry point established when the library was linked.

In a UNIX-based system, for example, 'mymod!func8' would direct the database manager to look for the library $inst_home_dir/sqllib/function/mymod and to use entry point func8 within that library.

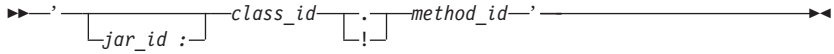
On Windows 32-bit operating systems, 'mymod!func8' would direct the database manager to load the mymod.dll file and call the func8() function in the dynamic link library (DLL).

If the string is not properly formed, an error (SQLSTATE 42878) is raised.

In any case, the body of every external function should be in a directory that is available on every partition of the database.

- For LANGUAGE JAVA:

  The *string* specified contains the optional jar file identifier, class identifier and method identifier, which the database manager invokes to execute the user-defined function being CREATEd. The class identifier and method identifier do not need to exist when the CREATE FUNCTION statement is performed. If a *jar_id* is specified, it must exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine.



  Extraneous blanks are not permitted within the single quotes.

  jar_id
  Identifies the jar identifier given to the jar collection when it was installed in the database. It can be either a simple identifier, or a schema qualified identifier. Examples are 'myJar' and 'mySchema.myJar'

  class_id
  Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix, for example, 'myPacks.UserFuncs'. The Java virtual machine will look in directory '.../myPacks/UserFuncs/' for the classes. On Windows 32-bit operating systems, the Java virtual machine will look in directory '...\myPacks\UserFuncs\'.

  method_id
  Identifies the method name of the Java object to be invoked.

- For LANGUAGE OLE:

  The *string* specified is the OLE programmatic identifier (progid) or class identifier (clsid), and method identifier, which the database manager invokes to execute the user-defined function being CREATEd. The programmatic identifier or class identifier, and

method identifier do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42724) is raised.

Extraneous blanks are not permitted within the single quotes.

progid
Identifies the programmatic identifier of the OLE object.

progid is not interpreted by the database manager but only forwarded to the OLE APIs at run time. The specified OLE object must be creatable and support late binding (also called IDispatch-based binding).

clsid
Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a *progid* in the case that an OLE object is not registered with a progid. The *clsid* has the form:

{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}

where 'n' is an alphanumeric character. *clsid* is not interpreted by the database manager but only forwarded to the OLE APIs at run time.

method_id
Identifies the method name of the OLE object to be invoked.

NAME *identifier*
This clause identifies the name of the user-written code which implements the function being defined. The *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the user-defined function body is written.

C This means the database manager will call the user-defined function as if it were a C function. The user-defined function must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA This means the database manager will call the user-defined function as a method in a Java class.

OLE This means the database manager will call the user-defined function as if it were a method exposed by an OLE automation object. The user-defined function must conform with the OLE automation data types and invocation mechanism, as described in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is only supported for user-defined functions stored in DB2 for Windows 32-bit operating systems.

For information on creating LANGUAGE OLE DB external table functions, see "CREATE FUNCTION (OLE DB External Table)".

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

DB2GENERAL
Used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a Java class. This can only be specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

SQL
Used to specify the conventions for passing parameters to and returning the value from external functions that conform to C language calling and linkage conventions. This must be specified when LANGUAGE C or LANGUAGE OLE is used.

DETERMINISTIC or **NOT DETERMINISTIC**
This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same table from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a NOT DETERMINISTIC table function would be a function that retrieves data from a data source such as a file.

FENCED or **NOT FENCED**
This clause specifies whether or not the function is considered "safe" to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a function is registered as FENCED, the database manager protects its internal resources (for example, data buffers) from access by the function.

Most functions will have the option of running as FENCED or NOT FENCED. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION:
Use of NOT FENCED for functions not adequately coded, reviewed and tested can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user defined functions are used.

Only FENCED can be specified for a function with LANGUAGE OLE or NOT THREADSAFE (SQLSTATE 42613).

If the function is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

Either SYSADM authority, DBADM authority, or a special authority (CREATE_NOT_FENCED_ROUTINE) is required to register a user-defined function as NOT FENCED.

THREADSAFE or **NOT THREADSAFE**
Specifies whether the function is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the function is defined with LANGUAGE other than OLE:

- If the function is defined as THREADSAFE, the database manager can invoke the function in the same process as other routines. In general, to be threadsafe, a function should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED functions can be THREADSAFE.
- If the function is defined as NOT THREADSAFE, the database manager will never invoke the function in the same process as another routine.

For FENCED functions, THREADSAFE is the default if the LANGUAGE is JAVA. For all other languages, NOT THREADSAFE is the default. If the function is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).

For NOT FENCED functions, THREADSAFE is the default. NOT THREADSAFE cannot be specified (SQLSTATE 42613).

RETURNS NULL ON NULL INPUT or **CALLED ON NULL INPUT**
This optional clause may be used to avoid a call to the external function if any of the arguments is null. If the user-defined function is defined to

have no parameters, then of course this null argument condition cannot arise, and it does not matter how this specification is coded.

If RETURNS NULL ON NULL INPUT is specified, and if, at table function OPEN time, any of the function's arguments are null, then the user-defined function is not called. The result of the attempted table function scan is the empty table (a table with no rows).

If CALLED ON NULL INPUT is specified, then regardless of whether any arguments are null, the user-defined function is called. It can return a null value or a normal (non-null) value. But responsibility for testing for null argument values lies with the UDF.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

NO SQL, CONTAINS SQL, READS SQL DATA
Indicates whether the function issues any SQL statements and, if so, what type.

NO SQL
Indicates that the function cannot execute any SQL statements (SQLSTATE 38001).

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the function (SQLSTATE 38004 or 42985). Statements that are not supported in any function return a different error (SQLSTATE 38003 or 42985).

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the function (SQLSTATE 38002 or 42985). Statements that are not supported in any function return a different error (SQLSTATE 38003 or 42985).

STATIC DISPATCH
This optional clause indicates that at function resolution time, DB2 chooses a function based on the static types (declared types) of the parameters of the function.

NO EXTERNAL ACTION or **EXTERNAL ACTION**
This optional clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. Optimizations that assume functions have no external impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

NO SCRATCHPAD or **SCRATCHPAD** *length*

This optional clause may be used to specify whether a scratchpad is to be provided for an external function. (It is strongly recommended that user-defined functions be re-entrant, so a scratchpad provides a means for the function to "save state" from one call to the next.)

If SCRATCHPAD is specified, then at first invocation of the user-defined function, memory is allocated for a scratchpad to be used by the external function. This scratchpad has the following characteristics:

- *length*, if specified, sets the size of the scratchpad in bytes and must be between 1 and 32 767 (SQLSTATE 42820). The default value is 100.
- It is initialized to all X'00''s.
- Its scope is the SQL statement. There is one scratchpad per reference to the external function in the SQL statement. So if the UDFX function in the following statement is defined with the SCRATCHPAD keyword, two scratchpads would be assigned.

```
SELECT A.C1, B.C2
  FROM TABLE (UDFX(:hv1)) AS A,
       TABLE (UDFX(:hv1)) AS B
    WHERE ...
```

- It is persistent. It is initialized at the beginning of the execution of the statement, and can be used by the external table function to preserve the state of the scratchpad from one call to the next. If the FINAL CALL keyword is also specified for the UDF, then the scratchpad is NEVER altered by DB2, and any resources anchored in the scratchpad should be released when the special FINAL call is made.

  If NO FINAL CALL is specified or defaulted, then the external table function should clean up any such resources on the CLOSE call, as DB2 will re-initialize the scratchpad on each OPEN call. This determination of FINAL CALL or NO FINAL CALL and the associated behavior of the scratchpad could be an important consideration, particularly if the table function will be used in a subquery or join, since that is when multiple OPEN calls can occur during the execution of a statement.
- It can be used as a central point for system resources (for example, memory) which the external function might acquire. The function could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

  (As outlined above, the FINAL CALL/NO FINAL CALL keyword is used to control the re-initialization of the scratchpad, and also dictates when the external table function should release resources anchored in the scratchpad.)

If SCRATCHPAD is specified, then on each invocation of the user-defined function an additional argument is passed to the external function which addresses the scratchpad.

If NO SCRATCHPAD is specified then no scratchpad is allocated or passed to the external function.

FINAL CALL or **NO FINAL CALL**
This optional clause specifies whether a final call (and a separate first call) is to be made to an external function. It also controls when the scratchpad is re-initialized. If NO FINAL CALL is specified, then DB2 can 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.

For external table functions, the call-type argument is ALWAYS present, regardless of which option is chosen.

If the final call is being made because of an interrupt or end-of-transaction, the UDF may not issue any SQL statements except for CLOSE cursor (SQLSTATE 38505). A special value is passed in the ″call type″ argument for these special final call situations.

DISALLOW PARALLEL
This clause specifies that, for a single reference to the function, the invocation of the function cannot be parallelized. Table functions are always run on a single partition.

NO DBINFO or **DBINFO**
This optional clause specifies whether certain specific information known by DB2 will be passed to the UDF as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613).

If DBINFO is specified, then a structure is passed to the UDF which contains the following information:

- Data base name - the name of the currently connected database.
- Application ID - unique application ID which is established for each connection to the database.
- Application Authorization ID - the application run-time authorization ID, regardless of the nested UDFs in between this UDF and the application.
- Code page - identifies the database code page.
- Schema name - not applicable to external table functions.
- Table name - not applicable to external table functions.
- Column name - not applicable to external table functions.
- Database version/release- identifies the version, release and modification level of the database server invoking the UDF.
- Platform - contains the server's platform type.

- Table function result column numbers - an array of the numbers of the table function result columns actually needed for the particular statement referencing the function. Only provided for table functions, it enables the UDF to optimize by only returning the required column values instead of all column values.

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 9 223 372 036 854 775 807 inclusive.

If the CARDINALITY clause is not specified for a table function, DB2 will assume a finite value as a default- the same value assumed for tables for which the RUNSTATS utility has not gathered statistics.

Warning: if a function does in fact have infinite cardinality, i.e. it returns a row every time it is called to do so, never returning the ″end-of-table″ condition, then queries which require the ″end-of-table″ condition to correctly function will be infinite, and will have to be interrupted. Examples of such queries are those involving GROUP BY and ORDER BY. The user is advised to not write such UDFs.

TRANSFORM GROUP *group-name*

Indicates the transform group to be used for user-defined structured type transformations when invoking the function. A transform is required if the function definition includes a user-defined structured type as a parameter data type. If this clause is not specified, the default group name DB2_FUNCTION is used. If the specified (or default) *group-name* is not defined for a referenced structured type, an error results (SQLSTATE 42741). If a required FROM SQL transform function is not defined for the given group-name and structured type, an error results (SQLSTATE 42744).

INHERIT SPECIAL REGISTERS

This optional clause specifies that updatable special registers in the function will inherit their initial values from the environment of the invoking statement. For a function invoked in the select-statement of a cursor, the initial values are inherited from the environment when the cursor is opened. For a routine invoked in a nested object (for example a trigger or view), the initial values are inherited from the runtime environment (not inherited from the object definition).

No changes to the special registers are passed back to the invoker of the function.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

NOT FEDERATED
This optional clause indicates that federated objects cannot be used in any SQL statement in the function. Using a federated object will result in an error (SQLSTATE 55047).

Notes:

- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values. For example, a constant which may be used as an input value could have a built-in data type that is different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:
  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- For portability of UDFs across platforms, it is recommended to use the following data types:
  - DOUBLE or REAL instead of FLOAT
  - DECIMAL instead of NUMERIC
  - CLOB (or BLOB) instead of LONG VARCHAR
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- Only routines defined as NO SQL can be used to define an index extension (SQLSTATE 428F8).
- ***Table access restrictions***

  If a function is defined as READS SQL DATA, no statement in the function can access a table that is being modified by the statement which invoked the function (SQLSTATE 57053). For example, suppose the user-defined function BONUS() is defined as READS SQL DATA. If the statement UPDATE EMPLOYEE SET SALARY = SALARY + BONUS(EMPNO) is invoked, no SQL statement in the BONUS function can read from the EMPLOYEE table.
- ***Compatibilities***
  - For compatibility with DB2 for z/OS and OS/390:
    - The following syntax is accepted as the default behavior:
      - ASUTIME NO LIMIT
      - NO COLLID

    - PROGRAM TYPE SUB
    - STAY RESIDENT NO
  - For compatibility with previous versions of DB2:
    - PARAMETER STYLE DB2SQL can be specified in place of PARAMETER STYLE SQL
    - NOT VARIANT can be specified in place of DETERMINISTIC
    - VARIANT can be specified in place of NOT DETERMINISTIC
    - NULL CALL can be specified in place of CALLED ON NULL INPUT
    - NOT NULL CALL can be specified in place of RETURNS NULL ON NULL INPUT
- ***Privileges***
  - The definer of a function always receives the EXECUTE privilege WITH GRANT OPTION on the function, as well as the right to drop the function.
  - When the function is used in an SQL statement, the function definer must have the EXECUTE privilege on any packages used by the function.

Examples:

Example 1: The following registers a table function written to return a row consisting of a single document identifier column for each known document in a text management system. The first parameter matches a given subject area and the second parameter contains a given string.

Within the context of a single session, the UDF will always return the same table, and therefore it is defined as DETERMINISTIC. Note the RETURNS clause which defines the output from DOCMATCH. FINAL CALL must be specified for each table function. In addition, the DISALLOW PARALLEL keyword is added as 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 DB2 optimizer.

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOC_ID CHAR(16))
  EXTERNAL NAME '/common/docfuncs/rajiv/udfmatch'
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  NOT FENCED
  SCRATCHPAD
  FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20
```

Example 2: The following registers an OLE table function that is used to retrieve message header information and the partial message text of messages in Microsoft Exchange.

```
CREATE FUNCTION MAIL()
  RETURNS TABLE (TIMERECEIVED DATE,
                 SUBJECT VARCHAR(15),
                 SIZE INTEGER,
                 TEXT VARCHAR(30))
  EXTERNAL NAME 'tfmail.header!list'
  LANGUAGE OLE
  PARAMETER STYLE SQL
  NOT DETERMINISTIC
  FENCED
  CALLED ON NULL INPUT
  SCRATCHPAD
  FINAL CALL
  NO SQL
  EXTERNAL ACTION
  DISALLOW PARALLEL
```

Related reference:

- "Basic predicate" in the *SQL Reference, Volume 1*
- "CREATE FUNCTION (OLE DB External Table)" on page 235
- "CREATE FUNCTION (SQL Scalar, Table or Row)" on page 254
- "CREATE FUNCTION (External Scalar)" on page 190
- "SQL statements allowed in routines" in the *SQL Reference, Volume 1*
- "Special registers" in the *SQL Reference, Volume 1*
- "Promotion of data types" in the *SQL Reference, Volume 1*

CREATE FUNCTION (OLE DB External Table)

This statement is used to register a user-defined OLE DB external table function to access data from an OLE DB provider.

A *table function* may be used in the FROM clause of a SELECT.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

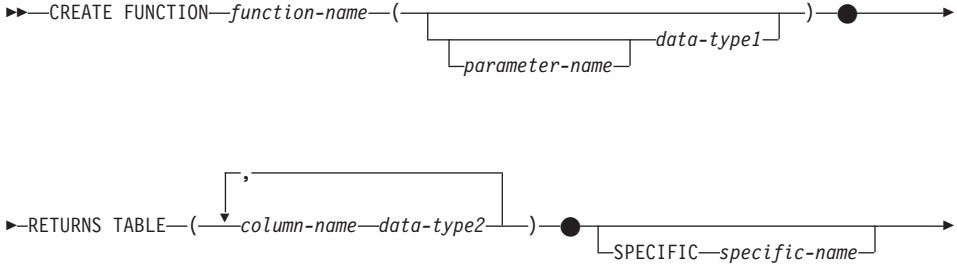
Authorization:

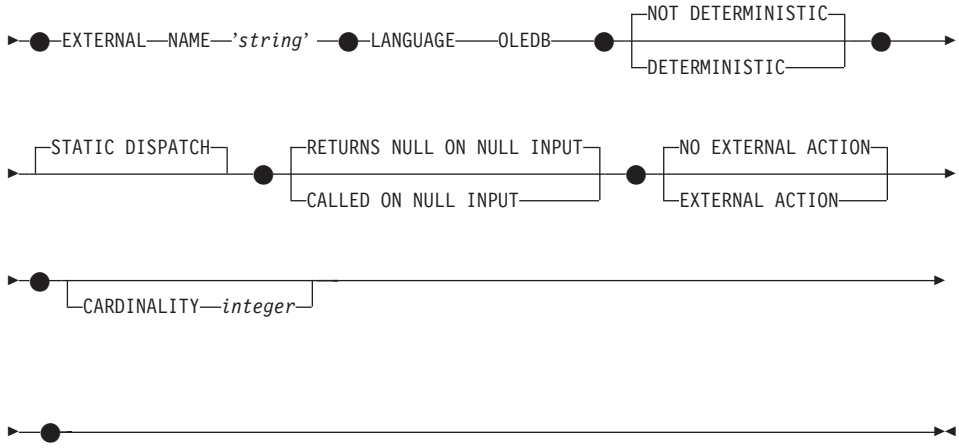
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATE_EXTERNAL_ROUTINE authority on the database, and at least one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
  - CREATEIN privilege on the schema, if the schema name of the function exists.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:





Description:

function-name

Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

The same name can be used for more than one function if there is some difference in the signature of the functions. Although there is no prohibition against it, an external user-defined table function should not be given the same name as a built-in function.

parameter-name
Specifies an optional name for the parameter.

data-type1
Identifies the input parameter of the function, and specifies the data type of the parameter. If no input parameter is specified, then data is retrieved from the external source possibly subsetted through query optimization. The input parameter can be any character or graphic string data type and it passes command text to an OLE DB provider.

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example,

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Length is not considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type. For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. A duplicate signature raises an SQL error (SQLSTATE 42723).

RETURNS TABLE
Specifies that the output of the function is a table. The parentheses that follow this keyword delimit a list of the names and types of the columns of the table, resembling the style of a simple CREATE TABLE statement which has no additional specifications (constraints, for example).

column-name
Specifies the name of the column which must be the same as the corresponding rowset column name. The name cannot be qualified and the same name cannot be used for more than one column of the table.

data-type2
Specifies the data type of the column.

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

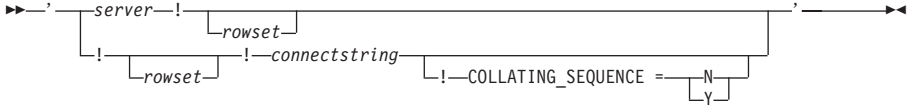
If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

EXTERNAL NAME *'string'*
This clause identifies the external table and an OLE DB provider.

The 'string' option is a string constant with a maximum of 254 characters.

The string specified is used to establish a connection and session with a OLE DB provider, and retrieve data from a rowset. The OLE DB provider and data source do not need to exist when the CREATE FUNCTION statement is performed.



server
Identifies the local name of a data source as defined by "CREATE SERVER".

rowset
Identifies the rowset (table) exposed by the OLE DB provider. Fully qualified table names must be provided for OLE DB providers that support catalog or schema names.

connectstring
String version of the initialization properties needed to connect to a data source. The basic format of a connection string is based on the ODBC connection string. The string contains a series of keyword/value pairs separated by semicolons. The equal sign (=) separates each keyword and its value. Keywords are the descriptions of the OLE DB initialization properties (property set DBPROPSET_DBINIT) or provider-specific keywords.

COLLATING_SEQUENCE
Specifies whether the data source uses the same collating sequence as DB2 Universal Database. For details, see "CREATE SERVER". Valid values are as follows:

Y = Same collating sequence

N = Different collating sequence

If COLLATING_SEQUENCE is not specified, then the data source is assumed to have a different collating sequence from DB2 Universal Database.

If *server* is provided, *connectstring* or COLLATING_SEQUENCE are not allowed in the external name. They are defined as server options CONNECTSTRING and COLLATING_SEQUENCE. If no *server* is provided, a *connectstring* must be provided. If *rowset* is not provided, the table function must have an input parameter to pass through command text to the OLE DB provider.

LANGUAGE OLEDB
This means the database manager will deploy a built-in generic OLE DB consumer to retrieve data from the OLE DB provider. No table function implementation is required by the developer.

LANGUAGE OLEDB table functions can be created on any platform, but only executed on platforms supported by Microsoft OLE DB.

DETERMINISTIC or **NOT DETERMINISTIC**
This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same table from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC.

STATIC DISPATCH
This optional clause indicates that at function resolution time, DB2 chooses a function based on the static types (declared types) of the parameters of the function.

RETURNS NULL ON NULL INPUT or **CALLED ON NULL INPUT**
This optional clause may be used to avoid a call to the external function if any of the arguments is null. If the user-defined function is defined to have no parameters, then of course this null argument condition cannot arise.

If RETURNS NULL ON NULL INPUT is specified and if at execution time any one of the function's arguments is null, the user-defined function is not called and the result is the empty table; that is, a table with no rows.

If CALLED ON NULL INPUT is specified, then at execution time regardless of whether any arguments are null, the user-defined function is called. It can return an empty table or not, depending on its logic. But responsibility for testing for null argument values lies with the UDF.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

NO EXTERNAL ACTION or **EXTERNAL ACTION**
This optional clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. Optimizations that assume functions with no external impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

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, DB2 will assume a finite value as a default- the same value assumed for tables for which the RUNSTATS utility has not gathered statistics.

Warning: if a function does in fact have infinite cardinality, i.e. it returns a row every time it is called to do so, never returning the ″end-of-table″ condition, then queries which require the ″end-of-table″ condition to correctly function will be infinite, and will have to be interrupted. Examples of such queries are those involving GROUP BY and ORDER BY. The user is advised to not write such UDFs.

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NOT VARIANT can be specified in place of DETERMINISTIC
    - VARIANT can be specified in place of NOT DETERMINISTIC
    - NULL CALL can be specified in place of CALLED ON NULL INPUT
    - NOT NULL CALL can be specified in place of RETURNS NULL ON NULL INPUT
- FENCED, FINAL CALL, SCRATCHPAD, PARAMETER STYLE SQL, DISALLOW PARALLEL, NO DBINFO, NOT THREADSAFE, and NO SQL are implicit in the statement and can be specified.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values. For example, a constant which may be used as an input value could have a built-in data type that is different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:

  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- For portability of UDFs across platforms, it is recommended to use the following data types:
  - DOUBLE or REAL instead of FLOAT
  - DECIMAL instead of NUMERIC
  - CLOB (or BLOB) instead of LONG VARCHAR
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- ***Privileges***

  The definer of a function always receives the EXECUTE privilege WITH GRANT OPTION on the function, as well as the right to drop the function.

Examples:

Example 1: The following registers an OLE DB table function, which retrieves order information from a Microsoft Access database. The connection string is defined in the external name.

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER,
                 customerid CHAR(5),
                 employeeid INTEGER,
                 orderdate TIMESTAMP,
                 requireddate TIMESTAMP,
                 shippeddate TIMESTAMP,
                 shipvia INTEGER,
                 freight dec(19,4))
  LANGUAGE OLEDB
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;
                         Data Source=c:\sqllib\samples\oledb\nwind.mdb
  !COLLATING_SEQUENCE=Y';
```

Example 2: The following registers an OLE DB table function, which retrieves customer information from an Oracle database. The connection string is provided through a server definition. The table name is fully qualified in the external name. The local user john is mapped to the remote user dave. Other users will use the guest user ID in the connection string.

```
CREATE SERVER spirit
  WRAPPER OLEDB
  OPTIONS (CONNECTSTRING 'Provider=MSDAORA;Persist Security Info=False;
                         User ID=guest;password=pwd;Locale Identifier=1033;
                         OLE DB Services=CLIENTCURSOR;Data Source=spirit');

CREATE USER MAPPING FOR john
  SERVER spirit
```

```
  OPTIONS (REMOTE_AUTHID 'dave', REMOTE_PASSWORD 'mypwd');

CREATE FUNCTION customers ()
  RETURNS TABLE (customer_id INTEGER,
                 name  VARCHAR(20),
                 address VARCHAR(20),
                 city VARCHAR(20),
                 state VARCHAR(5),
                 zip_code INTEGER)
  LANGUAGE OLEDB
  EXTERNAL NAME 'spirit!demo.customer';
```

Example 3: The following registers an OLE DB table function, which retrieves information about stores from a MS SQL Server 7.0 database. The connection string is provided in the external name. The table function has an input parameter to pass through command text to the OLE DB provider. The rowset name does not need to be specified in the external name. The query example passes in SQL statement text to retrieve the top three stores.

```
CREATE FUNCTION favorites (varchar(600))
  RETURNS TABLE (store_id CHAR (4),
                 name  VARCHAR (41),
                 sales INTEGER)
  SPECIFIC favorites
  LANGUAGE OLEDB
  EXTERNAL NAME '!!Provider=SQLOLEDB.1;Persist Security Info=False;
                 User ID=sa;Initial Catalog=pubs;Data Source=WALTZ;
                 Locale Identifier=1033;Use Procedure for Prepare=1;
                 Auto Translate=False;Packet Size=4096;Workstation ID=WALTZ;
                 OLE DB Services=CLIENTCURSOR;';

SELECT *
  FROM TABLE (favorites
             (' select top 3 sales.stor_id as store_id, ' ||
                  ' stores.stor_name as name, ' ||
                  ' sum(sales. qty) as sales ' ||
              ' from sales, stores ' ||
              ' where sales.stor_id = stores.stor_id ' ||
              ' group by sales.stor_id, stores.stor_name ' ||
              ' order by sum(sales.qty) desc ')) as f;
```

Related reference:

- "Basic predicate" in the *SQL Reference, Volume 1*
- "CREATE SERVER" on page 328
- "CREATE USER MAPPING" on page 461
- "CREATE WRAPPER" on page 479
- "CREATE FUNCTION (External Table)" on page 217
- "Promotion of data types" in the *SQL Reference, Volume 1*

CREATE FUNCTION (Sourced or Template)

This statement is used to:

- Register a user-defined function, based on another existing scalar or column function, with an application server.
- Register a function template with an application server that is designated as a federated server. A *function template* is a partial function that contains no executable code. The user creates it for the purpose of mapping it to a data source function. After the mapping is created, the user can specify the function template in queries submitted to the federated server. When such a query is processed, the federated server will invoke the data source function to which the template is mapped, and return values whose data types correspond to those in the RETURNS portion of the template's definition.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

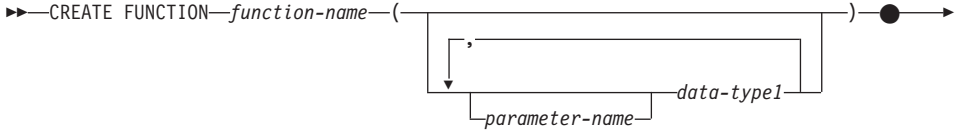
The privileges held by the authorization ID of the statement must include at least one of the following:

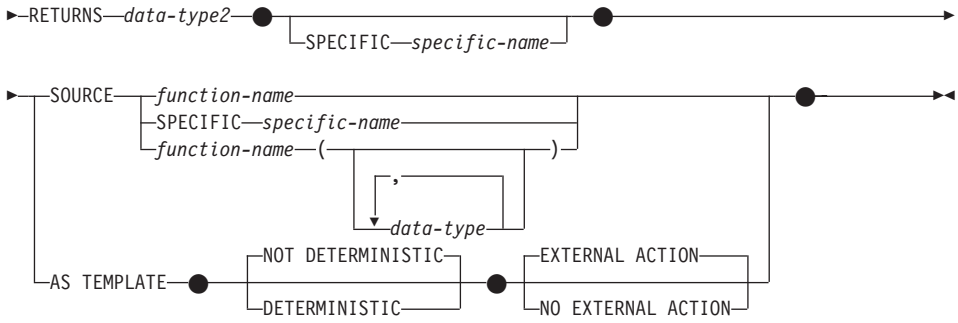
- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
- CREATEIN privilege on the schema, if the schema name of the function exists.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

No authority is required on a function referenced in the SOURCE clause.

Syntax:





Description:

function-name

Names the function or function template being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function or function template described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

When naming a user-defined function that is sourced on an existing function with the purpose of supporting the same function with a user-defined distinct type, the same name as the sourced function may be used. This allows users to use the same function with a user-defined distinct type without realizing that an additional definition was required. In general, the same name can be used for more than one function if there is some difference in the signature of the functions.

(data-type,...)
Identifies the number of input parameters of the function or function template, and specifies the data type of each parameter. One entry in the list must be specified for each parameter that the function or function template will expect to receive. No more than 90 parameters are allowed. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

It is possible to register a function or function template that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example,

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions or function templates within a schema are permitted to have exactly the same type for all corresponding parameters. (This restriction applies also to a function and function template within a schema that have the same name.) Lengths, precisions and scales are not considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL (4,3). For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...

CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

the second and fourth statements would fail because they are considered to be a duplicate functions.

parameter-name
Specifies an optional name for the parameter that is distinct from the names of all other parameters in this function.

data-type1
Specifies the data type of the parameter.

With a sourced scalar function any valid SQL data type may be used provided it is castable to the type of the corresponding parameter of the function identified in the SOURCE clause. A REF(*type-name*) data type cannot be specified as the data type of a parameter (SQLSTATE 42997).

Since the function is sourced, it is not necessary (but still permitted) to specify length, precision, or scale for the parameterized data types.

Instead, empty parentheses may be used (for example CHAR() may be used). A *parameterized data type* is any one of the data types that can be defined with a specific length, scale, or precision. The parameterized data types are the string data types and the decimal data types.

RETURNS

This mandatory clause identifies the output of the function or function template.

data-type2

Specifies the data type of the output.

Any valid SQL data type is valid, as is a distinct type, provided it is castable from the result type of the source function.

The parameter of a parameterized type need not be specified, as above for parameters of a sourced function. Instead, empty parentheses may be used, for example, VARCHAR().

For additional considerations and rules that apply to the specification of the data type in the RETURNS clause when the function is sourced on another, see the "Rules" section of this statement.

SPECIFIC *specific-name*

Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

SOURCE

Specifies that the function being created is to be implemented by another function (the source function) already known to the database manager. The source function can be either a built-in function (except for COALESCE, DBPARTITIONNUM, NULLIF, HASHEDVALUE, TYPE_ID, TYPE_NAME, TYPE_SCHEMA, or VALUE) or a previously created user-defined scalar function.

The SOURCE clause may be specified only for scalar or column functions; it may not be specified for table functions.

The SOURCE clause provides the identity of the other function.

function-name
Identifies the particular function that is to be used as the source and is valid only if there is exactly one specific function in the schema with this *function-name* for which the authorization ID of the statement has EXECUTE privilege. This syntax variant is not valid for a source function that is a built-in function.

If an unqualified name is provided, then the current SQL path (the value of the CURRENT PATH special register) is used to locate the function. The first schema in the function path that has a function with this name for which the authorization ID of the statement has EXECUTE privilege is selected.

If no function by this name exists in the named schema or if the name is not qualified and there is no function with this name in the function path, an error (SQLSTATE 42704) is raised. If there is more than one authorized specific instance of the function in the named or located schema, an error (SQLSTATE 42725) is raised. If a function by this name exists and the authorization ID of the statement does not have EXECUTE privilege on this function, an error (SQLSTATE 42501) is raised.

SPECIFIC *specific-name*
Identifies the particular user-defined function that is to be used as the source, by the *specific-name* either specified or defaulted to at function creation time. This syntax variant is not valid for a source function that is a built-in function.

If an unqualified name is provided, then the current SQL path is used to locate the function. The first schema in the function path that has a function with this specific name for which the authorization ID of the statement has EXECUTE privilege is selected.

If no function by this *specific-name* exists in the named schema or if the name is not qualified and there is no function with this *specific-name* in the SQL path, an error (SQLSTATE 42704) is raised. If a function by this *specific-name* exists, and the authorization ID of the statement does not have EXECUTE privilege on this function, an error (SQLSTATE 42501) is returned.

function-name (data-type,...)
Provides the function signature, which uniquely identifies the source function. This is the only valid syntax variant for a source function that is a built-in function.

The rules for function resolution are applied to select one function from the functions with the same function name, given the data types specified in the SOURCE clause. However, the data type of each parameter in the function selected must have the exact same type as the corresponding data type specified in the source function.

function-name
Gives the function name of the source function. If an unqualified name is provided, then the schemas of the user's SQL path are considered.

data-type
Must match the data type that was specified on the CREATE FUNCTION statement in the corresponding position (comma separated).

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match. For example, DECIMAL() will match a parameter whose data type was defined as DECIMAL(7,2)).

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement. This can be useful in assuring that the exact intended function will be used. Also note that synonyms for data types will be considered a match (for example DEC and NUMERIC will match).

A type of FLOAT(n) does not need to match the defined value for n since 0<n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

AS TEMPLATE
Indicates that this statement will be used to create a function template, not a function with executable code.

DETERMINISTIC or **NOT DETERMINISTIC**
This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same table from successive invocations with identical

inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC.

NOT DETERMINISTIC must be explicitly or implicitly specified if the body of the function accesses a special register or calls another non-deterministic function (SQLSTATE 428C2).

NO EXTERNAL ACTION or **EXTERNAL ACTION**
This optional clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. By specifying NO EXTERNAL ACTION, the system can use certain optimizations that assume functions have no external impacts.

EXTERNAL ACTION must be explicitly or implicitly specified if the body of the function calls another function that has an external action (SQLSTATE 428C2).

Rules:

- For convenience, in this section we will call the function being created CF and the function identified in the SOURCE clause SF, no matter which of the three allowable syntaxes was used to identify SF.
  - The unqualified name of CF and the unqualified name of SF can be different.
  - A function named as the source of another function can, itself, use another function as its source. Extreme care should be exercised when exploiting this facility because it could be very difficult to debug an application if an indirectly invoked function raises an error.
  - The following clauses are invalid if specified in conjunction with the SOURCE clause (because CF will inherit these attributes from SF):
    - CAST FROM ...,
    - EXTERNAL ...,
    - LANGUAGE ...,
    - PARAMETER STYLE ...,
    - DETERMINISTIC / NOT DETERMINISTIC,
    - FENCED / NOT FENCED,
    - RETURNS NULL ON NULL INPUT / CALLED ON NULL INPUT
    - EXTERNAL ACTION / NO EXTERNAL ACTION
    - NO SQL / CONTAINS SQL / READS SQL DATA
    - SCRATCHPAD / NO SCRATCHPAD
    - FINAL CALL / NO FINAL CALL
    - RETURNS TABLE (...)
    - CARDINALITY ...

- ALLOW PARALLEL / DISALLOW PARALLEL
- DBINFO / NO DBINFO
- THREADSAFE / NOT THREADSAFE
- INHERIT SPECIAL REGISTERS
- FEDERATED / NOT FEDERATED

An error (SQLSTATE 42613) will result from violation of these rules.

- The number of input parameters in CF must be the same as those in SF; otherwise an error (SQLSTATE 42624) is raised.
- It is not necessary for CF to specify length, precision, or scale for a parameterized data type in the case of:
  - The function's input parameters,
  - Its RETURNS parameter

  Instead, empty parentheses may be specified as part of the data type (for example: VARCHAR()) in order to indicate that the length/precision/scale will be the same as those of the source function, or determined by the casting.

  However, if length, precision, or scale is specified then the value in CF is checked against the corresponding value in SF as outlined below for input parameters and returns value.
- The specification of the input parameters of CF are checked against those of SF. The data type of each parameter of CF must either be the same as or be *castable* to the data type of the corresponding parameter of SF. If any parameter is not the same type or castable, an error (SQLSTATE 42879) is raised.

  Note that this rule provides no guarantee against an error occurring when CF is used. An argument that matches the data type and length or precision attributes of a CF parameter may not be assignable if the corresponding SF parameter has a shorter length or less precision. In general, parameters of CF should not have length or precision attributes that are greater than the attributes of the corresponding SF parameters.
- The specifications for the RETURNS data type of CF are checked against that of SF. The final RETURNS data type of SF, after any casting, must either be the same as or castable to the RETURNS data type of CF. Otherwise an error (SQLSTATE 42866) is raised.

  Note that this rule provides no guarantee against an error occurring when CF is used. A result value that matches the data type and length or precision attributes of the SF RETURNS data type may not be assignable if the CF RETURNS data type has a shorter length or less precision. Caution

should be used when choosing to specify the RETURNS data type of CF as having length or precision attributes that are less than the attributes of the SF RETURNS data type.

Notes:

- Determining whether one data type is castable to another data type does not consider length or precision and scale for parameterized data types such as CHAR and DECIMAL. Therefore, errors may occur when using a function as a result of attempting to cast a value of the source data type to a value of the target data type. For example, VARCHAR is castable to DATE but if the source type is actually defined as VARCHAR(5), an error will occur when using the function.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values (see "Promotion of data types"). For example, a constant which may be used as an input value could have a built-in data type different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:
  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- For a federated server to recognize a data source function, the function must map to a counterpart at the federated database. If the database contains no counterpart, the user must create the counterpart and then the mapping.

  The counterpart can be a function (scalar or source) or a function template. If the user creates a function and the required mapping, then, each time a query that specifies the function is processed, DB2 (1) compares strategies for invoking it with strategies for invoking the data source function, and (2) invokes the function that is expected to require less overhead.

  If the user creates a function template and the mapping, then each time a query that specifies the template is processed, DB2 invokes the data source function that it maps to, provided that an access plan for invoking this function exists.
- ***Privileges***

The definer of a function always receives the EXECUTE privilege on the function, as well as the right to drop the function. The definer of the function is also given the WITH GRANT OPTION if any one of the following is true:

- The source function is a built-in function.
- The definer of the function has EXECUTE WITH GRANT OPTION on the source function.
- The function is a template.

Examples:

Example 1: Some time after the creation of Pellow's original CENTRE external scalar function, another user wants to create a function based on it, except this function is intended to accept only integer arguments.

```
CREATE FUNCTION MYCENTRE (INTEGER, INTEGER)
  RETURNS FLOAT
  SOURCE PELLOW.CENTRE (INTEGER, FLOAT)
```

Example 2: A distinct type, HATSIZE, has been created based on the built-in INTEGER data type. It would be useful to have an AVG function to compute the average hat size of different departments. This is easily done as follows:

```
CREATE FUNCTION AVG (HATSIZE) RETURNS HATSIZE
  SOURCE SYSIBM.AVG (INTEGER)
```

The creation of the distinct type has generated the required cast function, allowing the cast from HATSIZE to INTEGER for the argument and from INTEGER to HATSIZE for the result of the function.

Example 3: In a federated system, a user wants to invoke an Oracle UDF that returns table statistics in the form of values with double-precision floating points. The federated server can recognize this function only if there is a mapping between the function and a federated database counterpart. But no such counterpart exists. The user decides to provide one in the form of a function template, and to assign this template to a schema called NOVA. The user uses the following code to register the template with the federated server.

```
CREATE FUNCTION NOVA.STATS (DOUBLE, DOUBLE)
  RETURNS DOUBLE
  AS TEMPLATE DETERMINISTIC NO EXTERNAL ACTION
```

Example 4: In a federated system, a user wants to invoke an Oracle UDF that returns the dollar amounts that employees of a particular organization earn as bonuses. The federated server can recognize this function only if there is a mapping between the function and a federated database counterpart. No such counterpart exists; thus, the user creates one in the form of a function template. The user uses the following code to register this template with the federated server.

```
CREATE FUNCTION BONUS ()
  RETURNS DECIMAL (8,2)
  AS TEMPLATE DETERMINISTIC NO EXTERNAL ACTION
```

Related reference:

- "Functions" in the *SQL Reference, Volume 1*
- "Basic predicate" in the *SQL Reference, Volume 1*
- "CREATE FUNCTION MAPPING" on page 263
- "Promotion of data types" in the *SQL Reference, Volume 1*
- "Casting between data types" in the *SQL Reference, Volume 1*

CREATE FUNCTION (SQL Scalar, Table or Row)

This statement is used to define a user-defined SQL scalar, table or row function. A *scalar function* returns a single value each time it is invoked, and is generally valid wherever an SQL expression is valid. A *table function* may be used in a FROM clause and returns a table. A *row function* may be used as a transform function and returns a row.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- For each table, view or nickname identified in any fullselect:
  - CONTROL privilege on that table, view, or nickname, or
  - SELECT privilege on that table, view, or nickname

  and at least one of the following:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the view does not exist
  - CREATEIN privilege on the schema, if the schema name of the view refers to an existing schema

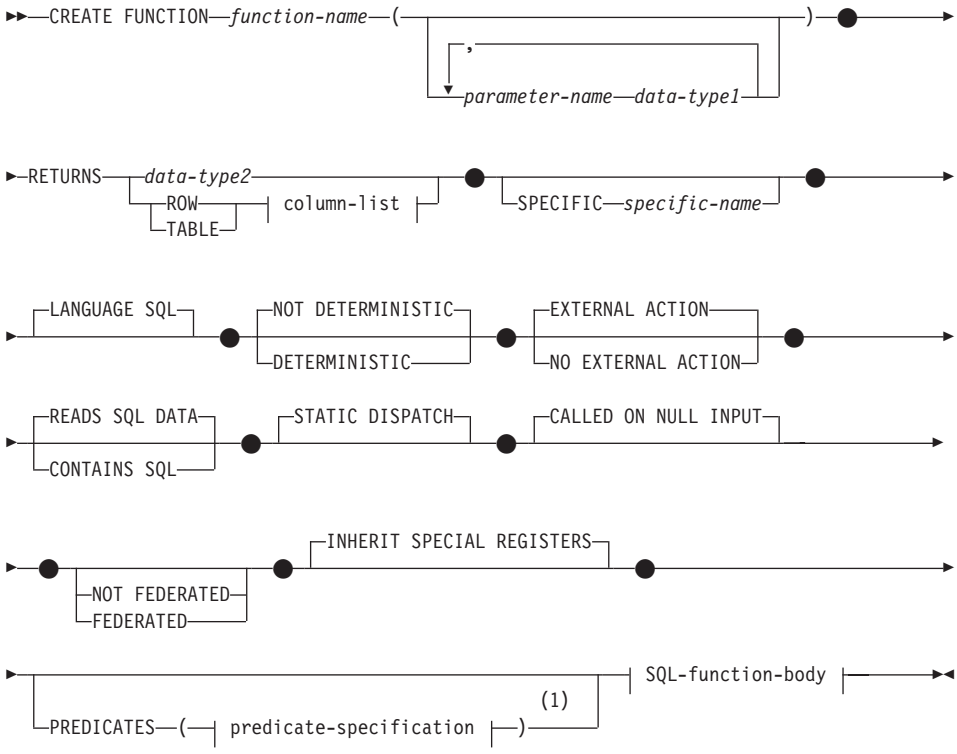
Group privileges other than PUBLIC are not considered for any table or view specified in the CREATE FUNCTION statement.

Authorization requirements of the data source for the table or view referenced by the nickname are applied when the function is invoked. The authorization ID of the connection may be mapped to a different remote authorization ID.

If a function definer can only create the function because the definer has SYSADM authority, the definer is granted implicit DBADM authority for the purpose of creating the function.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:



column-list:

SQL-function-body:

Notes:

1 Valid only if RETURNS specifies a scalar result (data-type2)

Description:

function-name
Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier

for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with "SYS" (SQLSTATE 42939).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

The same name can be used for more than one function if there is some difference in the signature of the functions. Although there is no prohibition against it, an external user-defined table function should not be given the same name as a built-in function.

parameter-name
A name that is distinct from the names of all other parameters in this function.

data-type1
Specifies the data type of the parameter:

- SQL data type specifications and abbreviations that may be specified in the *data-type1* definition of a CREATE TABLE statement.
- REF may be specified, but that REF is unscoped. The system does not attempt to infer the scope of the parameter or result. Inside the body of the function, a reference type can be used in a dereference operation only by first casting it to have a scope. Similarly, a reference returned by an SQL function can be used in a dereference operation only by first casting it to have a scope.
- LONG VARCHAR and LONG VARGRAPHIC data types may not be used (SQLSTATE 42815).

RETURNS
This mandatory clause identifies the type of output of the function.

data-type2
Specifies the data type of the output.

In this statement, exactly the same considerations apply as for the parameters of SQL functions described above under *data-type1* for function parameters.

ROW *column-list*
Specifies that the output of the function is a single row. If the function returns more than one row, an error is raised (SQLSTATE 21505). The *column-list* must include at least two columns (SQLSTATE 428F0).

A row function can only be used as a transform function for a structured type (having one structured type as its parameter and returning only base types).

TABLE *column-list*
Specifies that the output of the function is a table.

column-list
The list of column names and data types returned for a ROW or TABLE function

column-name
Specifies the name of this column. The name cannot be qualified and the same name cannot be used for more than one column of the row.

data-type3
Specifies the data type of the column, and can be any data type supported by a parameter of the SQL function.

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error is raised (SQLSTATE 42710).

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error is raised (SQLSTATE 42882).

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

LANGUAGE SQL
Specifies that the function is written using SQL.

DETERMINISTIC or **NOT DETERMINISTIC**
This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same table from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC.

NOT DETERMINISTIC must be explicitly or implicitly specified if the body of the function accesses a special register or calls another non-deterministic function (SQLSTATE 428C2).

NO EXTERNAL ACTION or **EXTERNAL ACTION**
This optional clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. By specifying NO EXTERNAL ACTION, the system can use certain optimizations that assume functions have no external impacts.

EXTERNAL ACTION must be explicitly or implicitly specified if the body of the function calls another function that has an external action (SQLSTATE 428C2).

READS SQL DATA or **CONTAINS SQL**
Indicates what type of SQL statements can be executed. Because the SQL statement supported is the RETURN statement, the distinction has to do with whether or not the expression is a subquery.

READS SQL DATA
Indicates that SQL statements that do not modify SQL data can be executed by the function (SQLSTATE 42985).

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the function (SQLSTATE 42985).

STATIC DISPATCH
This optional clause indicates that at function resolution time, DB2 chooses a function based on the static types (declared types) of the parameters of the function.

CALLED ON NULL INPUT
This clause indicates that the function is called regardless of whether any of its arguments are null. It can return a null value or a non-null value. Responsibility for testing null argument values lies with the user-defined function.

The phrase NULL CALL may be used in place of CALLED ON NULL INPUT.

FEDERATED or **NOT FEDERATED**
This optional clause specifies whether or not federated objects can be used. If neither option is specified, the body of the function is examined. If a federated object is referenced in the body of the function, a warning is returned (SQLSTATE 01639) and the function is created as FEDERATED. If no federated object is referenced, the function is defined as NOT FEDERATED.

If FEDERATED is specified, federated objects can be accessed from SQL statements in the function.

Statements within the function that access federated objects may be subject to special authorization rules.

If NOT FEDERATED is specified, federated objects cannot be used in any SQL statement in the function (SQLSTATE 429BA).

INHERIT SPECIAL REGISTERS
This optional clause indicates that updatable special registers in the function will inherit their initial values from the environment of the invoking statement. For a function that is invoked in the select-statement of a cursor, the initial values are inherited from the environment when the cursor is opened. For a routine that is invoked in a nested object (for example, a trigger or a view), the initial values are inherited from the runtime environment (not the object definition).

No changes to the special registers are passed back to the caller of the function.

Some special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore never inherited from the caller.

PREDICATES
For predicates using this function, this clause identifies those that can exploit the index extensions, and can use the optional SELECTIVITY clause for the predicate's search condition. If the PREDICATES clause is specified, the function must be defined as DETERMINISTIC with NO EXTERNAL ACTION (SQLSTATE 42613).

predicate-specification
For details on predicate specification, see "CREATE FUNCTION (External Scalar)".

SQL-function-body
Specifies the body of the function. Parameter names can be referenced in the SQL-function-body. Parameter names may be qualified with the function name to avoid ambiguous references.

If the SQL-function-body is a dynamic compound statement, it must contain at least one RETURN statement, and a RETURN statement must be executed when the function is called (SQLSTATE 42632). If the function is a table or row function, it can contain only one RETURN statement, which must be the last statement in the dynamic compound statement (SQLSTATE 429BD).

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NULL CALL can be specified in place of CALLED ON NULL INPUT
- Resolution of function calls inside the function body is done according to the function path that is effective for the CREATE FUNCTION statement and does not change after the function is created.
- If an SQL function contains multiple references to any of the date or time special registers, all references return the same value, and it will be the same value returned by the register invocation in the statement that called the function.
- The body of an SQL function cannot contain a recursive call to itself or to another function or method that calls it, since such a function could not exist to be called.
- The following rules are enforced by all statements that create functions or methods:
  - A function may not have the same signature as a method (comparing the first *parameter-type* of the function with the *subject-type* of the method).
  - A function and a method may not be in an overriding relationship. That is, if the function were a method with its first parameter as subject, it must not override, or be overridden by, another method. For more information about overriding methods, see the "CREATE TYPE (Structured)" statement.
  - Because overriding does not apply to functions, it is permissible for two functions to exist such that, if they were methods, one would override the other.

  For the purpose of comparing parameter-types in the above rules:
  - Parameter-names, lengths, AS LOCATOR, and FOR BIT DATA are ignored.
  - A subtype is considered to be different from its supertype.
- ***Table access restrictions***

  If a function is defined as READS SQL DATA, no statement in the function can access a table that is being modified by the statement which invoked the function (SQLSTATE 57053). For example, suppose the user-defined

function BONUS() is defined as READS SQL DATA. If the statement UPDATE EMPLOYEE SET SALARY = SALARY + BONUS(EMPNO) is invoked, no SQL statement in the BONUS function can read from the EMPLOYEE table.

- *Privileges*

  The definer of a function always receives the EXECUTE privilege on the function, as well as the right to drop the function. The definer of a function is also given the WITH GRANT OPTION on the function if the definer has WITH GRANT OPTION on all privileges required to define the function, or if the definer has SYSADM or DBADM authority.

  The definer of a function only acquires privileges if the privileges from which they are derived exist at the time the function is created. The definer must have these privileges either directly, or because PUBLIC has the privileges. Privileges held by groups of which the function definer is a member are not considered. When using the function, the connected user's authorization ID must have the valid privileges on the table or view that the nickname references at the data source.

Examples:

Example 1: Define a scalar function that returns the tangent of a value using the existing sine and cosine functions.

```
CREATE FUNCTION TAN (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(X)/COS(X)
```

Example 2: Define a transform function for the structured type PERSON.

```
CREATE FUNCTION FROMPERSON (P PERSON)
  RETURNS ROW (NAME VARCHAR(10), FIRSTNAME VARCHAR(10))
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN VALUES (P..NAME, P..FIRSTNAME)
```

Example 3: 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
  RETURN
    SELECT EMPNO, LASTNAME, FIRSTNME
      FROM EMPLOYEE
      WHERE EMPLOYEE.WORKDEPT = DEPTEMPLOYEES.DEPTNO
```

Example 4: Define a scalar function that reverses a string.

```
CREATE FUNCTION REVERSE(INSTR VARCHAR(4000))
  RETURNS VARCHAR(4000)
  DETERMINISTIC NO EXTERNAL ACTION CONTAINS SQL
  BEGIN ATOMIC
  DECLARE REVSTR, RESTSTR VARCHAR(4000) DEFAULT '';
  DECLARE LEN INT;
  IF INSTR IS NULL THEN
  RETURN NULL;
  END IF;
  SET (RESTSTR, LEN) = (INSTR, LENGTH(INSTR));
  WHILE LEN > 0 DO
  SET (REVSTR, RESTSTR, LEN)
    = (SUBSTR(RESTSTR, 1, 1) || REVSTR,
    SUBSTR(RESTSTR, 2, LEN - 1),
    LEN - 1);
  END WHILE;
  RETURN REVSTR;
END
```

Related reference:

- "Basic predicate" in the *SQL Reference, Volume 1*
- "CREATE TYPE (Structured)" on page 427
- "RETURN statement" on page 794
- "Compound SQL (Dynamic)" on page 123
- "CREATE FUNCTION (External Scalar)" on page 190
- "SQL statements allowed in routines" in the *SQL Reference, Volume 1*
- "Special registers" in the *SQL Reference, Volume 1*

CREATE FUNCTION MAPPING

The CREATE FUNCTION MAPPING statement is used to:

- Create a mapping between a federated database function or function template and a data source function. The mapping can associate the federated database function or template with a function at either (1) a specified data source or (2) a range of data sources; for example, all data sources of a particular type and version.
- Disable a default mapping between a federated database function and a data source function.

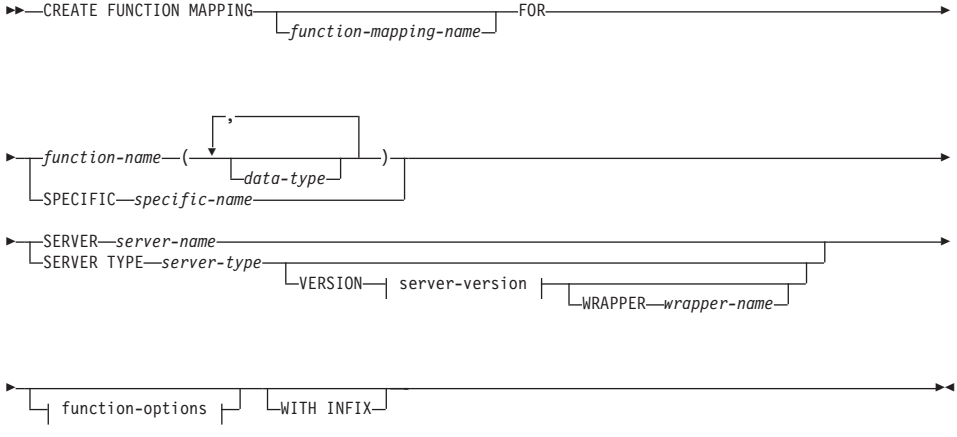
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

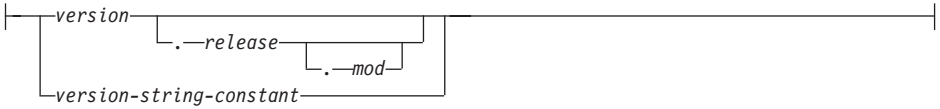
Authorization:

The authorization ID of the statement must have SYSADM or DBADM authority.

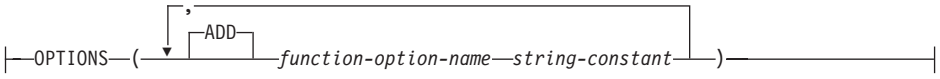
Syntax:



server-version:



function-options:



Description:

function-mapping-name
Names the function mapping. The name must not identify a function mapping that is already described in the catalog (SQLSTATE 42710).

If the *function-mapping-name* is omitted, a system-generated unique name is assigned.

function-name
Is the qualified or unqualified name of the function or function template to map from.

data-type
For a function or function template that has any input parameters, *data-type* specifies the data type of such a parameter. The *data type* cannot be LONG VARCHAR, LONG VARGRAPHIC, DATALINK, a large object (LOB) type, or a user-defined type.

SPECIFIC *specific-name*
Identifies the function or function template to map from. Specify *specific-name* if the function or function template does not have a unique *function-name* in the federated database.

SERVER *server-name*
Names the data source that contains the function that is being mapped to.

TYPE *server-type*
Identifies the type of data source that contains the function that is being mapped to.

VERSION
Identifies the version of the data source denoted by *server-type*.

version
Specifies the version number. *version* must be an integer.

release
Specifies the number of the release of the version denoted by *version*. *release* must be an integer.

mod
Specifies the number of the modification of the release denoted by *release*. *mod* must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release*, and, if applicable, *mod* (for example, '8.0.3').

WRAPPER *wrapper-name*
Specifies the name of the wrapper that the federated server uses to interact with data sources of the type and version denoted by *server-type* and *server-version*.

OPTIONS
Indicates what function mapping options are to be enabled.

ADD
Enables one or more function mapping options.

function-option-name
Names a function mapping option that applies either to the function mapping or to the data source function included in the mapping.

string-constant
Specifies the setting for *function-option-name* as a character string constant.

WITH INFIX
Specifies that the data source function be generated in infix format.

Notes:

- A federated database function or function template can map to a data source function if:
  - The federated database function or template has the same number of input parameters as the data source function.
  - The data types that are defined for the federated function or template are compatible with the corresponding data types that are defined for the data source function.
- If a distributed request references a DB2 function that maps to a data source function, the optimizer develops strategies for invoking either function when the request is processed. The DB2 function is invoked if

doing so requires less overhead than invoking the data source function. Otherwise, if invoking the DB2 function requires more overhead, then the data source function is invoked.

- If a distributed request references a DB2 function template that maps to a data source function, only the data source function can be invoked when the request is processed. The template cannot be invoked because it has no executable code.
- Default function mappings can be rendered inoperable by disabling them (they cannot be dropped). To disable a default function mapping, code the CREATE FUNCTION MAPPING statement so that it specifies the name of the DB2 function within the mapping and sets the DISABLE option to 'Y'.
- Functions in the SYSIBM schema do not have a specific name. To override the default function mapping for a function in the SYSIBM schema, specify *function-name* with qualifier SYSIBM and function name (such as LENGTH).
- A CREATE FUNCTION MAPPING statement within a given unit of work (UOW) cannot be processed under either of the following conditions:
  - The statement references a single data source, and the UOW already includes a SELECT statement that references a nickname for a table or view within this data source.
  - The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes a SELECT statement that references a nickname for a table or view within one of these data sources.

Examples:

Example 1: Map a function template to a UDF that all Oracle data sources can access. The template is called STATS and belongs to a schema called NOVA. The Oracle UDF is called STATISTICS and belongs to a schema called STAR.

```
CREATE FUNCTION MAPPING MY_ORACLE_FUN1
  FOR NOVA.STATS ( DOUBLE, DOUBLE )
  SERVER TYPE ORACLE
  OPTIONS ( REMOTE_NAME 'STAR.STATISTICS' )
```

Example 2: Map a function template called BONUS to a UDF, also called BONUS, that is used at an Oracle data source called ORACLE1.

```
CREATE FUNCTION MAPPING MY_ORACLE_FUN2
  FOR BONUS()
  SERVER ORACLE1
  OPTIONS ( REMOTE_NAME 'BONUS')
```

Example 3: Assume that there is a default function mapping between the WEEK system function that is defined to the federated database and a similar function that is defined to Oracle data sources. When a query that requests Oracle data and that references WEEK is processed, either WEEK or its Oracle

counterpart will be invoked, depending on which one is estimated by the optimizer to require less overhead. The DBA wants to find out how performance would be affected if only WEEK were invoked for such queries. To ensure that WEEK is invoked each time, the DBA must disable the mapping.

```
CREATE FUNCTION MAPPING
  FOR SYSFUN.WEEK(INT)
  TYPE ORACLE
  OPTIONS ( DISABLE 'Y' )
```

Example 4: Map the local function UCASE(CHAR) to a UDF that's used at an Oracle data source called ORACLE2. Include the estimated number of instructions per invocation of the Oracle UDF.

```
CREATE FUNCTION MAPPING MY_ORACLE_FUN4
  FOR SYSFUN.UCASE(CHAR)
  SERVER ORACLE2
  OPTIONS
     ( REMOTE_NAME 'UPPERCASE',
     INSTS_PER_INVOC '1000' )
```

Related reference:

- "Function mapping options for federated systems" in the *Federated Systems Guide*

CREATE INDEX

The CREATE INDEX statement is used to:

- Create an index on a DB2 table
- Create an index specification, metadata that indicates to the optimizer that a data source table has an index.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

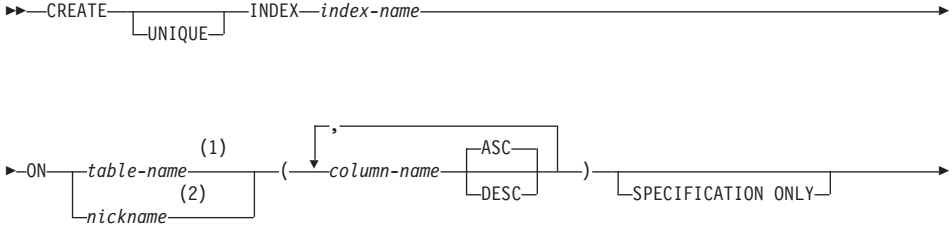
The privileges held by the authorization ID of the statement must include at least one of the following:

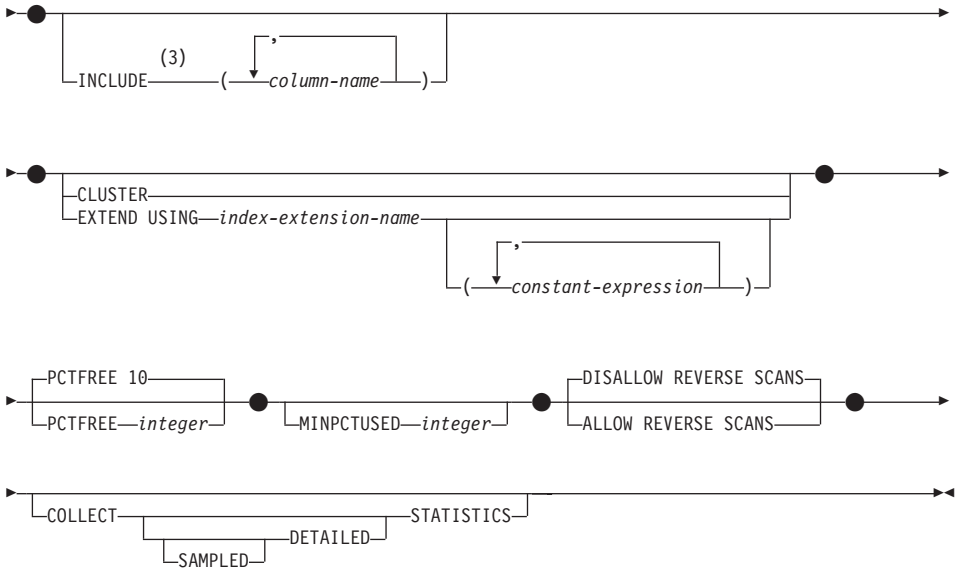
- SYSADM or DBADM authority.
- One of:
  - CONTROL privilege on the table
  - INDEX privilege on the table

  and one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the index does not exist
  - CREATEIN privilege on the schema, if the schema name of the index refers to an existing schema.

No explicit privilege is required to create an index on a declared temporary table.

Syntax:





Notes:

1 In a federated system, the *table-name* must identify a table in the federated database. It cannot identify a data source table.

2 If *nickname* is specified, the CREATE INDEX statement creates an index specification. INCLUDE, CLUSTER, PCTFREE, MINPCTUSED, DISALLOW REVERSE SCANS, ALLOW REVERSE SCANS, and COLLECT STATISTICS cannot be specified.

3 The INCLUDE clause may only be specified if UNIQUE is specified.

Description:

UNIQUE

If ON *table-name* is specified, UNIQUE prevents the table from containing two or more rows with the same value of the index key. The uniqueness is enforced at the end of the SQL statement that updates rows or inserts new rows.

The uniqueness 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.

When UNIQUE is used, null values are treated as any other values. For example, if the key is a single column that may contain null values, that column may contain no more than one null value.

If the UNIQUE option is specified, and the table has a partitioning key, the columns in the index key must be a superset of the partitioning key. That is, the columns specified for a unique index key must include all the columns of the partitioning key (SQLSTATE 42997).

Primary or unique keys cannot be subsets of dimensions (SQLSTATE 429BE).

If ON *nickname* is specified, UNIQUE should be specified only if the data for the index key contains unique values for every row of the data source table. The uniqueness will not be checked.

INDEX *index-name*
Names the index or index specification. The name, including the implicit or explicit qualifier, must not identify an index or index specification that is described in the catalog, or an existing index on a declared temporary table (SQLSTATE 42704). The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

The implicit or explicit qualifier for indexes on declared global temporary tables must be SESSION (SQLSTATE 428EK).

ON *table-name* **or** *nickname*
The *table-name* identifies a table on which an index is to be created. The table must be a base table (not a view), a materialized query table described in the catalog, or a declared temporary table. The name of a declared temporary table must be qualified with SESSION. The *table-name* must not identify a catalog table (SQLSTATE 42832). If UNIQUE is specified and *table-name* is a typed table, it must not be a subtable (SQLSTATE 429B3). If UNIQUE is specified, the *table-name* cannot be a materialized query table (SQLSTATE 42809).

nickname is the nickname on which an index specification is to be created. The *nickname* references either a data source table whose index is described by the index specification, or a data source view that is based on such a table. The *nickname* must be listed in the catalog.

column-name
For an index, *column-name* identifies a column that is to be part of the index key. For an index specification, *column-name* is the name by which the federated server references a column of a data source table.

Each *column-name* must be an unqualified name that identifies a column of the table. Up to 16 columns can be specified. If *table-name* is a typed table, up to 15 columns can be specified. If *table-name* is a subtable, at least one *column-name* must be introduced in the subtable; that is, not inherited from a supertable (SQLSTATE 428DS). No *column-name* can be repeated (SQLSTATE 42711).

The sum of the stored lengths of the specified columns must not be greater than 1024. If *table-name* is a typed table, the index key length limit is further reduced by 4 bytes.

Note that this length can be reduced by system overhead, which varies according to the data type of the column and whether it is nullable. For more information on overhead affecting this limit, see "Bytes Counts" in "CREATE TABLE".

No LOB column, DATALINK column, or distinct type column based on a LOB or DATALINK may be used as part of an index, even if the length attribute of the column is small enough to fit within the 1024-byte limit (SQLSTATE 54008). A structured type column can only be specified if the EXTEND USING clause is also specified (SQLSTATE 42962). If the EXTEND USING clause is specified, only one column can be specified, and the type of the column must be a structured type or a distinct type that is not based on a LOB, DATALINK, LONG VARCHAR, or LONG VARGRAPHIC (SQLSTATE 42997).

ASC
Specifies that index entries are to be kept in ascending order of the column values; this is the default setting. ASC cannot be specified for indexes that are defined with EXTEND USING (SQLSTATE 42601).

DESC
Specifies that index entries are to be kept in descending order of the column values. DESC cannot be specified for indexes that are defined with EXTEND USING (SQLSTATE 42601).

SPECIFICATION ONLY
Indicates that this statement will be used to create an index specification that applies to the data source table referenced by *nickname*. SPECIFICATION ONLY must be specified if *nickname* is specified (SQLSTATE 42601). It cannot be specified if *table-name* is specified (SQLSTATE 42601).

This clause cannot be used when creating an index on a declared temporary table (SQLSTATE 42995).

INCLUDE
This keyword introduces a clause that specifies additional columns to be appended to the set of index key columns. Any columns included with this clause are not used to enforce uniqueness. These included columns may improve the performance of some queries through index only access. The columns must be distinct from the columns used to enforce uniqueness (SQLSTATE 42711). The limits for the number of columns and sum of the length attributes apply to all of the columns in the unique key and in the index.

This clause cannot be used with declared temporary tables (SQLSTATE 42995).

column-name
Identifies a column that is included in the index but not part of the unique index key. The same rules apply as defined for columns of the unique index key. The keywords ASC or DESC may be specified following the column-name but have no effect on the order.

INCLUDE cannot be specified for indexes that are defined with EXTEND USING, or if *nickname* is specified (SQLSTATE 42601).

CLUSTER
Specifies that the index is the clustering index of the table. The cluster factor of a clustering index is maintained or improved dynamically as data is inserted into the associated table, by attempting to insert new rows physically close to the rows for which the key values of this index are in the same range. Only one clustering index may exist for a table so CLUSTER may not be specified if it was used in the definition of any existing index on the table (SQLSTATE 55012). A clustering index may not be created on a table that is defined to use append mode (SQLSTATE 428D8).

CLUSTER is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with declared temporary tables (SQLSTATE 42995).

EXTEND USING *index-extension-name*
Names the *index-extension* used to manage this index. If this clause is specified, then there must be only one *column-name* specified and that column must be a structured type or a distinct type (SQLSTATE 42997). The *index-extension-name* must name an index extension described in the catalog (SQLSTATE 42704). For a distinct type, the column must exactly match the type of the corresponding source key parameter in the index extension. For a structured type column, the type of the corresponding source key parameter must be the same type or a supertype of the column type (SQLSTATE 428E0).

This clause cannot be used with declared temporary tables (SQLSTATE 42995).

constant-expression
Identifies values for any required arguments for the index extension. Each expression must be a constant value with a data type that exactly matches the defined data type of the corresponding index extension parameters, including length or precision, and scale (SQLSTATE 428E0).

PCTFREE *integer*
Specifies what percentage of each index page to leave as free space when

building the index. The first entry in a page is added without restriction. When additional entries are placed in an index page at least *integer* percent of free space is left on each page. The value of *integer* can range from 0 to 99. However, if a value greater than 10 is specified, only 10 percent free space will be left in non-leaf pages. The default is 10.

PCTFREE is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with declared temporary tables (SQLSTATE 42995).

MINPCTUSED *integer*
Indicates whether index leaf pages are merged online, and the threshold for the minimum percentage of space used on an index leaf page. If, after a key is removed from an index leaf page, the percentage of space used on the page is at or below *integer* percent, an attempt is made to merge the remaining keys on this page with those of a neighboring page. If there is sufficient space on one of these pages, the merge is performed and one of the pages is deleted. The value of *integer* can be from 0 to 99. However, a value of 50 or below is recommended for performance reasons. Specifying this option will have an impact on update and delete performance. For type 2 indexes, merging is only done during update and delete operations when there is an exclusive table lock. If an exclusive table lock does not exist, keys are marked as pseudo deleted during update and delete operations, and no merging is done. Consider using the CLEANUP ONLY ALL option of REORG INDEXES to merge leaf pages instead of using the MINPCTUSED option of CREATE INDEX.

MINPCTUSED is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with declared temporary tables (SQLSTATE 42995).

DISALLOW REVERSE SCANS
Specifies that an index only supports forward scans or scanning of the index in the order defined at INDEX CREATE time. This is the default.

DISALLOW REVERSE SCANS is disallowed if *nickname* is specified (SQLSTATE 42601).

ALLOW REVERSE SCANS
Specifies that an index can support both forward and reverse scans; that is, in the order defined at INDEX CREATE time and in the opposite (or reverse) order.

ALLOW REVERSE SCANS is disallowed if *nickname* is specified (SQLSTATE 42601).

COLLECT STATISTICS
Specifies that basic index statistics are to be collected during index creation.

DETAILED
Specifies that extended index statistics (CLUSTERFACTOR and PAGE_FETCH_PAIRS) are also to be collected during index creation.

SAMPLED
Specifies that sampling can be used when compiling extended index statistics.

Rules:

- The CREATE INDEX statement will fail (SQLSTATE 01550) if attempting to create an index that matches an existing index. Two index descriptions are considered duplicates if:
  - the set of columns (both key and include columns) and their order in the index is the same as that of an existing index AND
  - the ordering attributes are the same AND
  - both the previously existing index and the one being created are non-unique OR the previously existing index is unique AND
  - if both the previously existing index and the one being created are unique, the key columns of the index being created are the same or a superset of key columns of the previously existing index.

Notes:

- *Compatibilities*
  - For compatibility with DB2 for OS/390:
    - The following syntax is tolerated and ignored:
      - CLOSE
      - DEFINE
      - FREEPAGE
      - gbpcache-block
      - PIECESIZE
      - TYPE 2
      - using-block
    - The following syntax is accepted as the default behavior:
      - COPY NO
      - DEFER NO
- Concurrent read/write access to the table is permitted while an index is being created. Once the index has been built, changes that were made to the table during index creation time are forward-fitted to the new index. Write access to the table is then briefly blocked while index creation completes, after which the new index becomes available.

To circumvent this default behavior, use the LOCK TABLE statement to explicitly lock the table before issuing a CREATE INDEX statement. (The table can be locked in either SHARE or EXCLUSIVE mode, depending on whether read access is to be allowed.)

- If the named table already contains data, CREATE INDEX creates the index entries for it. If the table does not yet contain data, CREATE INDEX creates a description of the index; the index entries are created when data is inserted into the table.
- Once the index is created and data is loaded into the table, it is advisable to issue the RUNSTATS command. The RUNSTATS command updates statistics collected on the database tables, columns, and indexes. These statistics are used to determine the optimal access path to the tables. By issuing the RUNSTATS command, the database manager can determine the characteristics of the new index. If data has been loaded before the CREATE INDEX statement is issued, it is recommended that the COLLECT STATISTICS option on the CREATE INDEX statement be used as an alternative to the RUNSTATS command.
- Creating an index with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- The optimizer can recommend indexes prior to creating the actual index.
- If an index specification is being defined for a data source table that has an index, the name of the index specification does not have to match the name of the index.
- The optimizer uses index specifications to improve access to the data source tables that the specifications apply to.
- The COLLECT STATISTICS options are not supported with declared temporary tables (SQLSTATE 42995).
- The COLLECT STATISTICS options are not supported if a nickname is specified (SQLSTATE 42601).

Examples:

Example 1: Create an index named UNIQUE_NAM on the PROJECT table. The purpose of the index is to ensure that there are not two entries in the table with the same value for project name (PROJNAME). The index entries are to be in ascending order.

```
CREATE UNIQUE INDEX UNIQUE_NAM
   ON PROJECT(PROJNAME)
```

Example 2: Create an index named JOB_BY_DPT on the EMPLOYEE table. Arrange the index entries in ascending order by job title (JOB) within each department (WORKDEPT).

```
CREATE INDEX JOB_BY_DPT
   ON EMPLOYEE (WORKDEPT, JOB)
```

Example 3: The nickname EMPLOYEE references a data source table called CURRENT_EMP. After this nickname was created, an index was defined on CURRENT_EMP. The columns chosen for the index key were WORKDEBT and JOB. Create an index specification that describes this index. Through this specification, the optimizer will know that the index exists and what its key is. With this information, the optimizer can improve its strategy to access the table.

```
CREATE UNIQUE INDEX JOB_BY_DEPT
  ON EMPLOYEE (WORKDEPT, JOB)
  SPECIFICATION ONLY
```

Example 4: Create an extended index type named SPATIAL_INDEX on a structured type column location. The description in index extension GRID_EXTENSION is used to maintain SPATIAL_INDEX. The literal is given to GRID_EXTENSION to create the index grid size.

```
CREATE INDEX SPATIAL_INDEX ON CUSTOMER (LOCATION)
  EXTEND USING (GRID_EXTENSION (x'00010010001000100040001 0'))
```

Example 5: Create an index named IDX1 on a table named TAB1, and collect basic index statistics on index IDX1.

```
CREATE INDEX IDX1 ON TAB1 (col1) COLLECT STATISTICS
```

Example 6: Create an index named IDX2 on a table named TAB1, and collect detailed index statistics on index IDX2.

```
CREATE INDEX IDX2 ON TAB1 (col2) COLLECT DETAILED STATISTICS
```

Example 7: Create an index named IDX3 on a table named TAB1, and collect detailed index statistics on index IDX3 using sampling.

```
CREATE INDEX IDX3 ON TAB1 (col3) COLLECT SAMPLED DETAILED STATISTICS
```

Related concepts:

- "Index specifications" in the *Federated Systems Guide*

Related reference:

- "CREATE TABLE" on page 332
- "Interaction of triggers and constraints" in the *SQL Reference, Volume 1*
- "CREATE INDEX EXTENSION" on page 277

Related samples:

- "dbstat.sqb -- Reorganize table and run statistics (MF COBOL)"
- "TbGenCol.java -- How to use generated columns (JDBC)"

CREATE INDEX EXTENSION

The CREATE INDEX EXTENSION statement creates an extension object for use with indexes on tables that have structured type or distinct type columns.

Invocation:

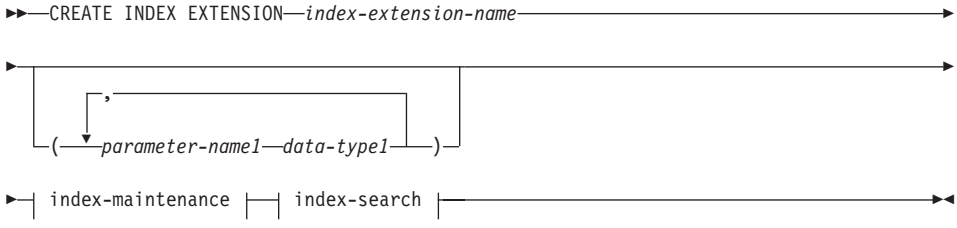
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

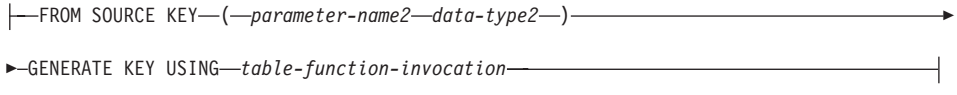
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the database (if the schema name of the index extension does not refer to an existing schema)
- CREATEIN privilege on the schema (if the schema name of the index extension refers to an existing schema)

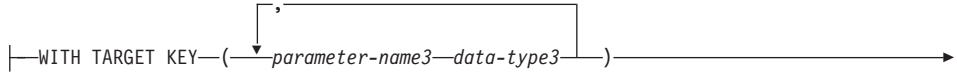
Syntax:



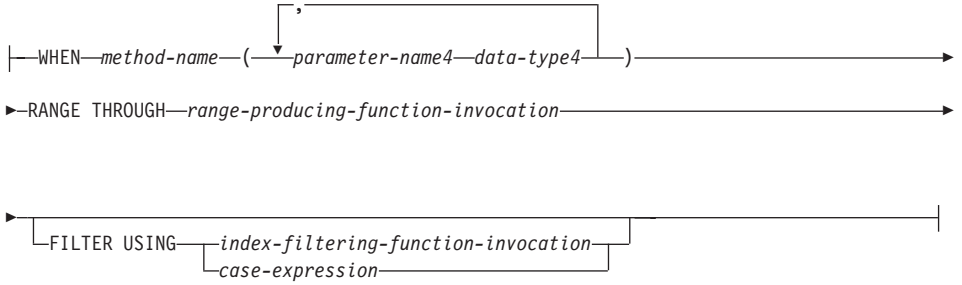
index-maintenance:



index-search:



search-method-definition:



Description:

index-extension-name
Names the index extension. The name, including the implicit or explicit qualifier, must not identify an index extension described in the catalog. If a two-part *index-extension-name* is specified, the schema name cannot begin with "SYS"; otherwise, an error (SQLSTATE 42939) is returned.

parameter-name1
Identifies a parameter that is passed to the index extension at CREATE INDEX time to define the actual behavior of this index extension. The parameter that is passed to the index extension is called an *instance parameter*, because that value defines a new instance of an index extension.

parameter-name1 must be unique within the definition of the index extension. No more than 90 parameters are allowed. If this limit is exceeded, an error (SQLSTATE 54023) is returned.

data-type1
Specifies the data type of each parameter. One entry in the list must be specified for each parameter that the index extension will expect to receive. The only SQL data types that may be specified are those that can be used as constants, such as VARCHAR, INTEGER, DECIMAL, DOUBLE, or VARGRAPHIC (SQLSTATE 429B5). The parameter value that is received by the index extension at CREATE INDEX must match *data-type1* exactly, including length, precision and scale (SQLSTATE 428E0).

index-maintenance
Specifies how the index keys of a structured or distinct type column are

maintained. Index maintenance is the process of transforming the source column to a target key. The transformation process is defined using a table function that has previously been defined in the database.

FROM SOURCE KEY (*parameter-name2 data-type2***)**
Specifies a structured data type or distinct type for the source key column that is supported by this index extension.

parameter-name2
Identifies the parameter that is associated with the source key column. A source key column is the index key column (defined in the CREATE INDEX statement) with the same data type as *data-type2*.

data-type2
Specifies the data type for *parameter-name2*. *data-type2* must be a user-defined structured type or a distinct type that is not sourced on LOB, DATALINK, LONG VARCHAR, or LONG VARGRAPHIC (SQLSTATE 42997). When the index extension is associated with the index at CREATE INDEX time, the data type of the index key column must:

- exactly match *data-type2* if it is a distinct type; or
- be the same type or a subtype of *data-type2* if it is a structured type

Otherwise, an error is returned (SQLSTATE 428E0).

GENERATE KEY USING *table-function-invocation*
Specifies how the index key is generated using a user-defined table function. Multiple index entries may be generated for a single source key data value. An index entry cannot be duplicated from a single source key data value (SQLSTATE 22526). The function can use *parameter-name1*, *parameter-name2*, or a constant as arguments. If the data type of *parameter-name2* is a structured data type, only the observer methods of that structured type can be used in its arguments (SQLSTATE 428E3). The output of the GENERATE KEY function must be specified in the TARGET KEY specification. The output of the function can also be used as input for the index filtering function specified on the FILTER USING clause.

The function used in *table-function-invocation* must:

- Resolve to a table function (SQLSTATE 428E4)
- Not be defined with LANGUAGE SQL (SQLSTATE 428E4)
- Not be defined with NOT DETERMINISTIC (SQLSTATE 428E4) or EXTERNAL ACTION (SQLSTATE 428E4)
- Be defined with NO SQL (SQLSTATE 428E4)

- Not have a structured data type, LOB, DATALINK, LONG VARCHAR, or LONG VARGRAPHIC (SQLSTATE 428E3) in the data type of the parameters, with the exception of system generated observer methods.
- Not include a subquery (SQLSTATE 428E3).
- Return columns with data types that follow the restrictions for data types of columns of an index defined without the EXTEND USING clause.

If an argument invokes another operation or routine, it must be an observer method (SQLSTATE 428E3).

The definer of the index extension must have EXECUTE privilege on this function.

index-search
Specifies how searching is performed by providing a mapping of the search arguments to search ranges.

WITH TARGET KEY
Specifies the target key parameters that are the output of the key generation function specified on the GENERATE KEY USING clause.

parameter-name3
Identifies the parameter associated with a given target key. *parameter-name3* corresponds to the columns of the RETURNS table as specified in the table function of the GENERATE KEY USING clause. The number of parameters specified must match the number of columns returned by that table function (SQLSTATE 428E2).

data-type3
Specifies the data type for each corresponding *parameter-name3*. *data-type3* must exactly match the data type of each corresponding output column of the RETURNS table, as specified in the table function of the GENERATE KEY USING clause (SQLSTATE 428E2), including the length, precision, and type.

SEARCH METHODS
Introduces the search methods that are defined for the index.

search-method-definition
Specifies the method details of the index search. It consists of a method name, the search arguments, a range producing function, and an optional index filter function.

WHEN *method-name*
The name of a search method. This is an SQL identifier that relates to the method name specified in the index exploitation rule (found in the

PREDICATES clause of a user-defined function). A *search-method-name* can be referenced by only one WHEN clause in the search method definition (SQLSTATE 42713).

parameter-name4
Identifies the parameter of a search argument. These names are for use in the RANGE THROUGH and FILTER USING clauses.

data-type4
The data type associated with a search parameter.

RANGE THROUGH *range-producing-function-invocation*
Specifies an external table function that produces search ranges. This function uses *parameter-name1*, *parameter-name4*, or a constant as arguments and returns a set of search ranges.

The table function used in *range-producing-function-invocation* must:

- Resolve to a table function (SQLSTATE 428E4)
- Not include a subquery (SQLSTATE 428E3) or SQL function (SQLSTATE 428E4) in its arguments
- Not be defined with LANGUAGE SQL (SQLSTATE 428E4)
- Not be defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 428E4)
- Be defined with NO SQL (SQLSTATE 428E4)
- The number and types of this function's results must relate to the results of the table function specified in the GENERATE KEY USING clause as follows (SQLSTATE 428E1):
  - Return up to twice as many columns as returned by the key transformation function
  - Have an even number of columns, in which the first half of the return columns define the start of the range (start key values), and the second half of the return columns define the end of the range (stop key values)
  - Have each start key column with the same type as the corresponding stop key column
  - Have the type of each start key column the same as the corresponding key transformation function column.

More precisely, let a_1:t_1, ..., a_n:t_n be the function result columns and data types of the key transformation function. The function result columns of the *range-producing-function-invocation* must be b_1:t_1, ..., b_m:t_m, c_1:t_1, ..., c_m:t_m, where m <= n and the "b" columns are the start key columns and the "c" columns are the stop key columns.

When the *range-producing-function-invocation* returns a null value as the start or stop key value, the semantics are undefined.

The definer of the index extension must have EXECUTE privilege on this function.

FILTER USING

Allows specification of an external function or a case expression to be used for filtering index entries that were returned after applying the range-producing function.

index-filtering-function-invocation

Specifies an external function to be used for filtering index entries. This function uses the *parameter-name1*, *parameter-name3*, *parameter-name4*, or a constant as arguments (SQLSTATE 42703) and returns an integer (SQLSTATE 428E4). If the value returned is 1, the row corresponding to the index entry is retrieved from the table. Otherwise, the index entry is not considered for further processing.

If not specified, index filtering is not performed.

The function used in the *index-filtering-function-invocation* must:

- Not be defined with LANGUAGE SQL (SQLSTATE 429B4)
- Not be defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 42845)
- Be defined with NO SQL (SQLSTATE 428E4)
- Not have a structured data type in the data type of any of the parameters (SQLSTATE 428E3).
- Not include a subquery (SQLSTATE 428E3)

If an argument invokes another function or method, these four rules are also enforced for this nested function or method. However, system generated observer methods are allowed as arguments to the filter function (or any function or method used as an argument), as long as the argument results in a built-in data type.

The definer of the index extension must have EXECUTE privilege on this function.

case-expression

Specifies a case expression for filtering index entries. Either *parameter-name1*, *parameter-name3*, *parameter-name4*, or a constant (SQLSTATE 42703) can be used in the *searched-when-clause* and *simple-when-clause*. An external function with the rules specified in FILTER USING *index-filtering-function-invocation* may be used in *result-expression*. Any function referenced in the *case-expression* must also conform to the four rules listed under *index-filtering-function-invocation*. In addition, subqueries cannot be used anywhere else in

the *case-expression* (SQLSTATE 428E4). The case expression must return an integer (SQLSTATE 428E4). A return value of 1 in the *result-expression* means the index entry is kept, otherwise the index entry is discarded.

Notes:

- Creating an index extension with a schema name that does not already exist will result in the implicit creation of that schema, provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

Examples:

Example 1: The following creates an index extension called *grid_extension* that uses a structured type SHAPE column in a table function called *gridEntry* to generate seven index target keys. This index extension also provides two index search methods to produce search ranges when given a search argument.

```
CREATE INDEX EXTENSION GRID_EXTENSION (LEVELS VARCHAR(20) FOR BIT DATA)
  FROM SOURCE KEY (SHAPECOL SHAPE)
  GENERATE KEY USING GRIDENTRY(SHAPECOL..MBR..XMIN,
                               SHAPECOL..MBR..YMIN,
                               SHAPECOL..MBR..XMAX,
                               SHAPECOL..MBR..YMAX,
                               LEVELS)
  WITH TARGET KEY (LEVEL INT, GX INT, GY INT,
                   XMIN INT, YMIN INT, XMAX INT, YMAX INT)
  SEARCH METHODS
  WHEN SEARCHFIRSTBYSECOND (SEARCHARG SHAPE)
  RANGE THROUGH GRIDRANGE(SEARCHARG..MBR..XMIN,
                               SEARCHARG..MBR..YMIN,
                               SEARCHARG..MBR..XMAX,
                               SEARCHARG..MBR..YMAX,
                               LEVELS)
  FILTER USING
    CASE WHEN (SEARCHARG..MBR..YMIN > YMAX) OR
       SEARCHARG..MBR..YMAX < YMIN) THEN 0
    ELSE CHECKDUPLICATE(LEVEL, GX, GY,
                               XMIN, YMIN, XMAX, YMAX,
                               SEARCHARG..MBR..XMIN,
                               SEARCHARG..MBR..YMIN,
                               SEARCHARG..MBR..XMAX,
                               SEARCHARG..MBR..YMAX,
                               LEVELS)
    END
  WHEN SEARCHSECONDBYFIRST (SEARCHARG SHAPE)
  RANGE THROUGH GRIDRANGE(SEARCHARG..MBR..XMIN,
                               SEARCHARG..MBR..YMIN,
                               SEARCHARG..MBR..XMAX,
                               SEARCHARG..MBR..YMAX,
```

```
                                LEVELS)
  FILTER USING
    CASE WHEN (SEARCHARG..MBR..YMIN > YMAX) OR
       SEARCHARG..MBR..YMAX < YMIN) THEN 0
    ELSE MBROVERLAP(XMIN, YMIN, XMAX, YMAX,
                                SEARCHARG..MBR..XMIN,
                                SEARCHARG..MBR..YMIN,
                                SEARCHARG..MBR..XMAX,
                                SEARCHARG..MBR..YMAX)
    END
```

Related reference:

- "Constants" in the *SQL Reference, Volume 1*

CREATE METHOD

This statement is used to associate a method body with a method specification that is already part of the definition of a user-defined structured type.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATEIN privilege on the schema of the structured type referred to in the CREATE METHOD statement
- The DEFINER of the structured type referred to in the CREATE METHOD statement.

To associate an external method body with its method specification, the authorization ID of the statement must also include at least one of the following:

- SYSADM or DBADM authority
- CREATE_EXTERNAL_ROUTINE authority on the database.

When creating an SQL method, the privileges held by the authorization ID of the statement must also include, for each table, view, or nickname identified in any fullselect:

- CONTROL privilege on that table, view, or nickname, or
- SELECT privilege on that table, view, or nickname

and at least one of the following:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the view does not exist
- CREATEIN privilege on the schema, if the schema name of the view refers to an existing schema.

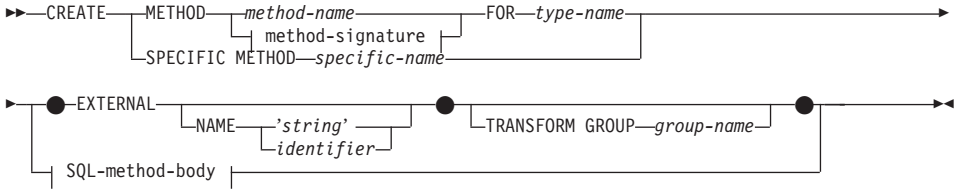
If the definer of an SQL method can only create the method because the definer has SYSADM authority, the definer is granted implicit DBADM authority for the purpose of creating the method.

Group privileges other than PUBLIC are not considered for any table or view specified in the CREATE METHOD statement.

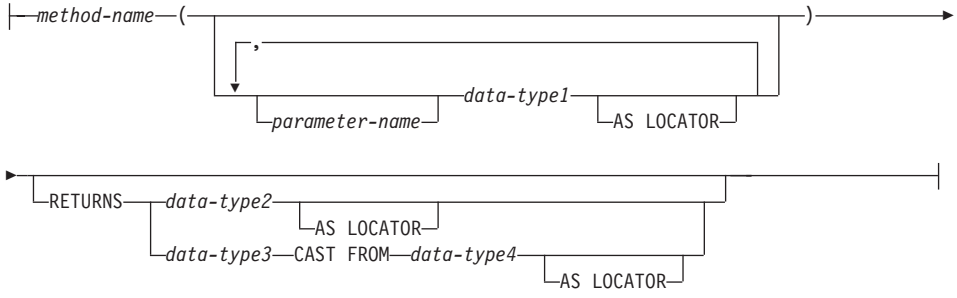
Authorization requirements of the data source for the table or view referenced by the nickname are applied when the method is invoked. The authorization ID of the connection may be mapped to a different remote authorization ID.

If the authorization ID has insufficient authority to perform the operation, an error is raised (SQLSTATE 42502).

Syntax:



method-signature:



SQL-method-body:

Description:

METHOD

Identifies an existing method specification that is associated with a user-defined structured type. The method-specification can be identified through one of the following means:

method-name

Names the method specification for which a method body is being defined. The implicit schema is the schema of the subject type

(*type-name*). There must be only one method specification for *type-name* that has this *method-name* (SQLSTATE 42725).

method-signature
Provides the method signature which uniquely identifies the method to be defined. The method signature must match the method specification that was provided on the CREATE TYPE or ALTER TYPE statement (SQLSTATE 42883).

method-name
Names the method specification for which a method body is being defined. The implicit schema is the schema of the subject type (*type-name*).

parameter-name
Identifies the parameter name. If parameter names are provided in the method signature, they must be exactly the same as the corresponding parts of the matching method specification. Parameter names are supported in this statement solely for documentation purposes.

data-type1
Specifies the data type of each parameter.

AS LOCATOR
For the LOB types or distinct types which are based on a LOB type, the AS LOCATOR clause can be added.

RETURNS
This clause identifies the output of the method. If a RETURNS clause is provided in the method signature, it must be exactly the same as the corresponding part of the matching method specification on CREATE TYPE. The RETURNS clause is supported in this statement solely for documentation purposes.

data-type2
Specifies the data type of the output.

AS LOCATOR
For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be returned by the method instead of the actual value.

data-type3 **CAST FROM** *data-type4*
This form of the RETURNS clause is used to return a different data type to the invoking statement from the data type that was returned by the function code.

AS LOCATOR
For LOB types or distinct types which are based on LOB

types, the AS LOCATOR clause can be used to indicate that a LOB locator is to be returned from the method instead of the actual value.

FOR *type-name*
Names the type for which the specified method is to be associated. The name must identify a type already described in the catalog. (SQLSTATE 42704) In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

SPECIFIC METHOD *specific-name*
Identifies the particular method, using the specific name either specified or defaulted to at CREATE TYPE time. The specific-name must identify a method specification in the named or implicit schema; otherwise, an error is raised (SQLSTATE 42704).

EXTERNAL
This clause indicates that the CREATE METHOD statement is being used to register a method, based on code written in an external programming language, and adhering to the documented linkage conventions and interface. The matching method-specification in CREATE TYPE must specify a LANGUAGE other than SQL. When the method is invoked, the subject of the method is passed to the implementation as an implicit first parameter.

If the NAME clause is not specified, "NAME *method-name*" is assumed.

NAME
This clause identifies the name of the user-written code which implements the method being defined.

'*string*'
The 'string' option is a string constant with a maximum of 254 characters. The format used for the string is dependent on the LANGUAGE specified. For more information on the specific language conventions, see "CREATE FUNCTION (External Scalar)".

identifier
This identifier specified is an SQL identifier. The SQL identifier is used as the library-id in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C (as defined in the method-specification on CREATE TYPE).

TRANSFORM GROUP *group-name*

Indicates the transform group that is used for user-defined structured type transformations when invoking the method. A transform is required since the method definition includes a user-defined structured type.

It is strongly recommended that a transform group name be specified; if this clause is not specified, the default group-name used is DB2_FUNCTION. If the specified (or default) group-name is not defined for a referenced structured type, an error results (SQLSTATE 42741). Likewise, if a required FROM SQL or TO SQL transform function is not defined for the given group-name and structured type, an error results (SQLSTATE 42744).

SQL-method-body

The SQL-method-body defines how the method is implemented if the method specification in CREATE TYPE is LANGUAGE SQL.

The SQL-method-body must comply with the following parts of method specification:

- DETERMINISTIC or NOT DETERMINISTIC (SQLSTATE 428C2)
- EXTERNAL ACTION or NO EXTERNAL ACTION (SQLSTATE 428C2)
- CONTAINS SQL or READS SQL DATA (SQLSTATE 42985)
- FEDERATED or NOT FEDERATED (SQLSTATE 429BA)

Parameter names can be referenced in the SQL-method-body. The subject of the method is passed to the method implementation as an implicit first parameter named SELF.

For additional details, see "Compound SQL (Dynamic)" and "RETURN Statement".

Rules:

- The method specification must be previously defined using the CREATE TYPE or ALTER TYPE statement before CREATE METHOD can be used (SQLSTATE 42723).
- If the method being created is an overriding method, those packages that are dependent on the following methods are invalidated:
  - The original method
  - Other overriding methods that have as their subject a supertype of the method being created

Notes:

- ***Privileges***

  The definer of a method always receives the EXECUTE privilege on the method, as well as the right to drop the method.

If an EXTERNAL method is created, the definer of the method always receives the EXECUTE privilege WITH GRANT OPTION.

If an SQL method is created, the definer of the method will only be given the EXECUTE privilege WITH GRANT OPTION on the method when the definer has WITH GRANT OPTION on all privileges required to define the method, or if the definer has SYSADM or DBADM authority. The definer of an SQL method only acquires privileges if the privileges from which they are derived exist at the time the method is created. The definer must have these privileges either directly, or because PUBLIC has the privileges. Privileges held by groups of which the method definer is a member are not considered. When using the method, the connected user's authorization ID must have the valid privileges on the table or view that the nickname references at the data source.

- ***Table access restrictions***

  If a method is defined as READS SQL DATA, no statement in the method can access a table that is being modified by the statement which invoked the method (SQLSTATE 57053).

Examples:

Example 1:

```
CREATE METHOD BONUS (RATE DOUBLE)
  FOR EMP
  RETURN SELF..SALARY * RATE
```

Example 2:

```
CREATE METHOD SAMEZIP (addr address_t)
  RETURNS INTEGER
  FOR address_t
  RETURN
    (CASE
      WHEN (self..zip = addr..zip)
        THEN 1
      ELSE 0
    END)
```

Example 3:

```
CREATE METHOD DISTANCE (address_t)
  FOR address_t
  EXTERNAL NAME 'addresslib!distance'
  TRANSFORM GROUP func_group
```

Related reference:

- "RETURN statement" on page 794
- "Compound SQL (Dynamic)" on page 123
- "CREATE FUNCTION (External Scalar)" on page 190

CREATE NICKNAME

The CREATE NICKNAME statement creates a nickname for a data source object, such as a table or a view.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

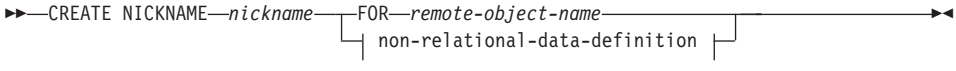
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

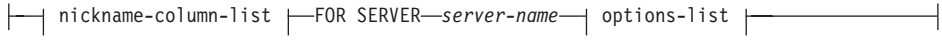
- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the federated database, if the implicit or explicit schema name of the nickname does not exist
- CREATEIN privilege on the schema, if the schema name of the nickname exists

In addition, the user's authorization ID at the data source must hold the privilege to select from the data source catalog the metadata about the table or view for which the nickname is being created.

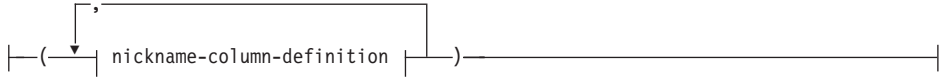
Syntax:



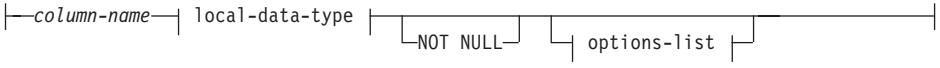
non-relational-data-definition:



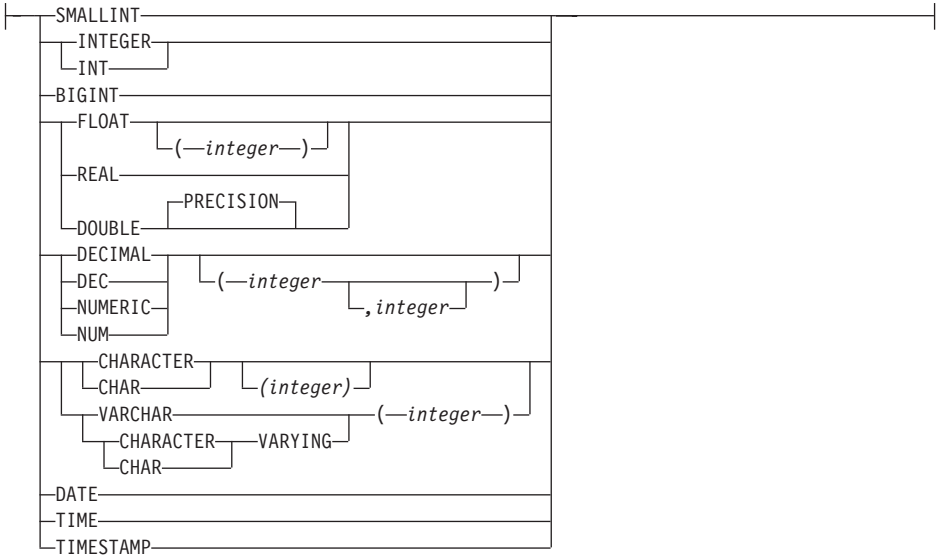
nickname-column-list:



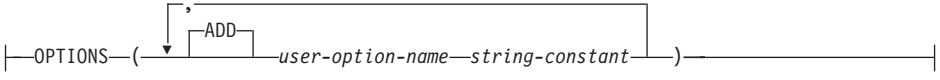
nickname-column-definition:



local-data-type:



options-list:



Description:

nickname
Names the nickname. The nickname specifies the federated server's identifier for the object at the data source. The nickname, including the implicit or explicit qualifier, must not identify a table, view, alias, or nickname described in the catalog. The schema name must not begin with 'SYS' (SQLSTATE 42939).

remote-object-name
Specifies a three-part identifier with format *data-source-name.remote-schema-name.remote-table-name*, or a two-part identifier with format *data-source-name.remote-table-name*, for use with data sources that do not support schema names.

data-source-name
Names the data source that contains the table or view for which the

nickname is being created. The *data-source-name* is the same name that was assigned to the data source in the CREATE SERVER statement.

remote-schema-name
Names the schema to which the table or view belongs. If the remote schema name contains any special or lowercase characters, it must be enclosed by double quotation marks.

remote-table-name
Names the specific data source object (such as a table or a view) for which the nickname is being created. The table cannot be a declared temporary table (SQLSTATE 42995). If the remote table name contains any special or lowercase characters, it must be enclosed by double quotation marks.

non-relational-data-definition
Defines the data to be accessed through a non-relational wrapper.

nickname-column-definition
Defines the local attributes of the column for the nickname. Some wrappers require these attributes to be specified, while other wrappers allow the attributes to be determined from the data source.

column-name
Specifies the local name for the column. The name may be different than the corresponding column of the *remote-object-name*.

local-data-type
Specifies the local data type for the column. Wrappers may only support a subset of the SQL data types. For descriptions of specific data types, see "CREATE TABLE".

NOT NULL
Specifies that the column does not allow null values.

options-list
Allows the specification of column options. Some wrappers may require column options to be specified.

FOR SERVER *server-name*
Specifies a server that was registered using the CREATE SERVER statement. This server will be used to access the data for the nickname.

options-list
Allows the specification of nickname options. For more information on the *user-option-name* and *string-constant* values that are supported, see "User options for federated systems".

Notes:

- The data source object (such as a table or a view) that the nickname references must already exist at the data source denoted by the first qualifier in *remote-object-name*.
- The federated server does not support those data source data types that correspond to the following DB2 data types: LONG VARCHAR, LONG VARGRAPHIC, DATALINK, and user-defined types. When a nickname is defined for a data source table or view, only those columns in the table or view that have supported data types will be defined to, and can be queried from, the federated database. When the CREATE NICKNAME statement is run against a table or view that has columns with unsupported data types, an error is issued.
- Because data types might be incompatible between data sources, the federated server makes minor adjustments to store remote catalog data locally as needed.
- The maximum allowable length of DB2 index names is 18 characters. If a nickname is being created for a table that has an index whose name exceeds this length, the entire name is not cataloged. Rather, DB2 truncates it to 18 characters. If the string formed by these characters is not unique within the schema to which the index belongs, DB2 attempts to make it unique by replacing the last character with 0. If the result is still not unique, DB2 changes the last character to 1. DB2 repeats this process with numbers 2 through 9, and if necessary, with numbers 0 through 9 for the name's seventeenth character, sixteenth character, and so on, until a unique name is generated. To illustrate: The index of a data source table is named ABCDEFGHIJKLMNOPQRSTUVWXYZ. The names ABCDEFGHIJKLMNOPQR and ABCDEFGHIJKLMNOPQ0 already exist in the schema to which this index belongs. The new name is over 18 characters; therefore, DB2 truncates it to ABCDEFGHIJKLMNOPQR. Because this name already exists in the schema, DB2 changes the truncated version to ABCDEFGHIJKLMNOPQ0. Because this latter name exists, too, DB2 changes the truncated version to ABCDEFGHIJKLMNOPQ1. This name does not already exist in the schema, so DB2 now accepts it as a new name.
- When a nickname is created for a data source object, DB2 stores the names of the nickname columns in the catalog. When the data source object is a table or a view, DB2 makes the nickname column names the same as the table or view column names. If a name exceeds the maximum allowable length for DB2 column names, DB2 truncates the name to this length. If the truncated version is not unique among the other column names in the table or view, DB2 makes it unique by following the procedure described in the preceding paragraph.
- If the definition of the remote data source object is changed (e.g., a column is deleted or a data type is changed), the nickname should be dropped and recreated. Otherwise, errors may occur when the nickname is used in an SQL statement.

Examples:

Example 1: Create a nickname for a view, DEPARTMENT, that is in a schema called HEDGES. This view is stored in a DB2 Universal Database for OS/390 and z/OS data source called OS390A.

```
CREATE NICKNAME DEPT FOR OS390A.HEDGES.DEPARTMENT
```

Example 2: Select all records from the view for which a nickname was created in Example 1. The view must be referenced by its nickname. (It can be referenced it by its own name only in pass-through sessions.)

```
SELECT * FROM OS390A.HEDGES.DEPARTMENT   Invalid
SELECT * FROM DEPT                       Valid after nickname DEPT is created
```

Example 3: The following example shows a CREATE NICKNAME statement for the table-structured file DRUGDATA1.TXT.

```
CREATE NICKNAME DRUGDATA1(DCODE INTEGER,DRUG CHAR(20),MANUFACTURER CHAR(20))
  FOR SERVER biochem_lab
  OPTIONS
    (FILE_PATH '/usr/pat/DRUGDATA1.TXT',
    COLUMN_DELIMITER ',',
    KEY_COLUMN 'Dcode',
    VALIDATE_DATA_FILE 'Y')
```

Related reference:

- "CREATE TABLE" on page 332
- "CREATE SERVER" on page 328
- "Column options for federated systems" in the *Federated Systems Guide*
- "User options for federated systems" in the *Federated Systems Guide*

CREATE PROCEDURE

The CREATE PROCEDURE statement defines a procedure with an application server.

There are two different types of procedures that can be created using this statement. Each of these is described separately.

- External. The procedure body is written in a programming language. The external executable is referenced by a procedure defined with an application server, along with various attributes of the procedure.
- SQL. The procedure body is written in SQL. The procedure body is defined with an application server along with various attributes of the procedure.

Related reference:

- "CREATE PROCEDURE (External)" on page 297
- "CREATE PROCEDURE (SQL)" on page 311

CREATE PROCEDURE (External)

The CREATE PROCEDURE (External) statement is used to define an external procedure with an application server.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATE_EXTERNAL_ROUTINE authority on the database and at least one of:
  - IMPLICIT_SCHEMA authority on the database, if the schema name of the procedure does not refer to an existing schema
  - CREATEIN privilege on the schema, if the schema name of the procedure refers to an existing schema

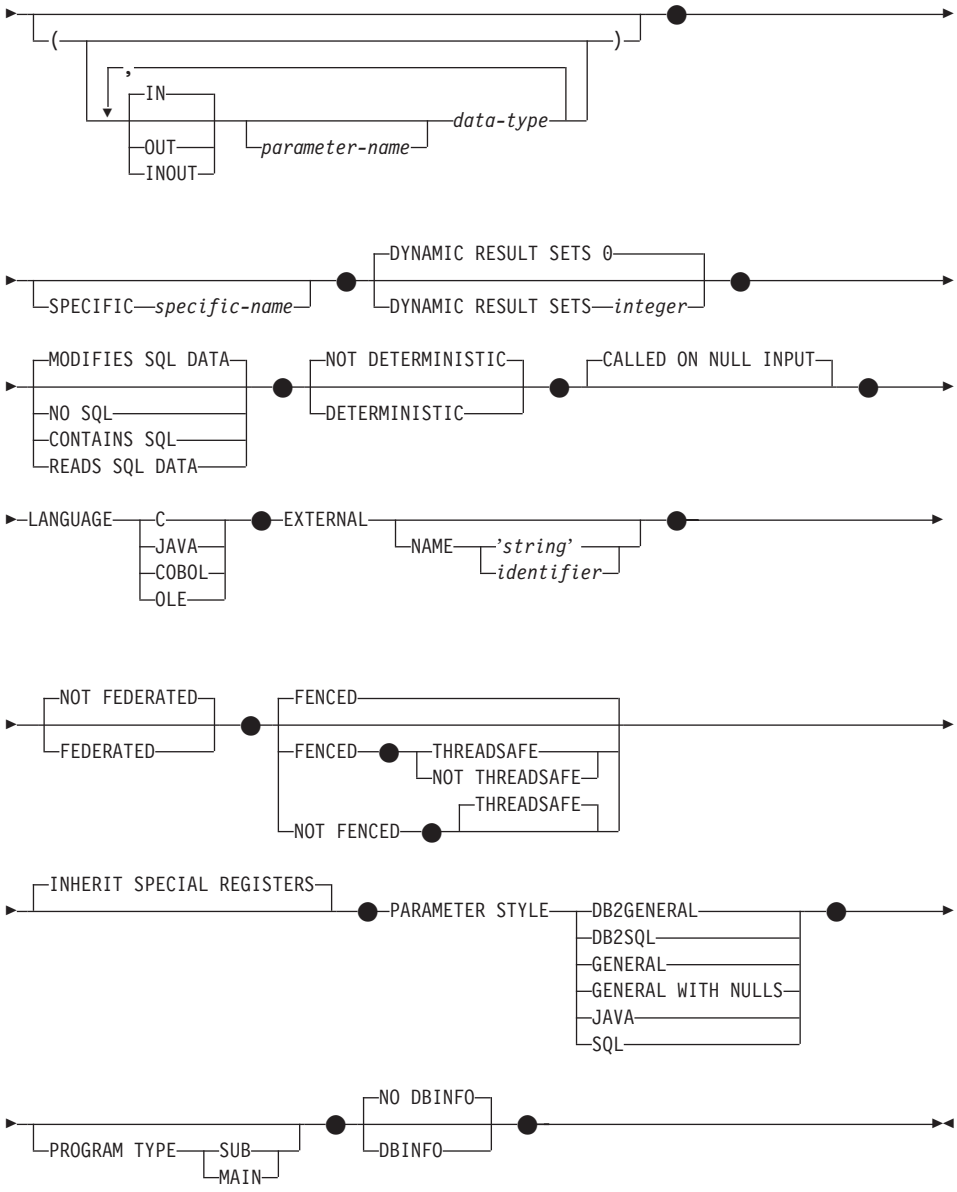
To create a not-fenced stored procedure, the privileges held by the authorization ID of the statement must also include at least one of the following:

- CREATE_NOT_FENCED_ROUTINE authority on the database
- SYSADM or DBADM authority.

To create a fenced stored procedure, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:



Description:

procedure-name

Names the procedure being defined. It is a qualified or unqualified name that designates a procedure. The unqualified form of *procedure-name* is an SQL identifier (with a maximum length of 128). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER

precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters must not identify a procedure described in the catalog (SQLSTATE 42723). The unqualified name, together with the number of the parameters, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

*(***IN** | **OUT** | **INOUT** *parameter-name data-type,...)*
Identifies the parameters of the procedure, and specifies the mode, data type, and optional name of each parameter. One entry in the list must be specified for each parameter that the procedure will expect.

No two identically-named procedures within a schema are permitted to have exactly the same number of parameters. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE PROCEDURE PART (IN NUMBER INT, OUT PART_NAME CHAR(35)) ...
CREATE PROCEDURE PART (IN COST DECIMAL(5,3), OUT COUNT INT) ...
```

the second statement will fail because the number of parameters in the procedure is the same, even if the data types are not.

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 for the procedure.

INOUT
Identifies the parameter as both an input and output parameter for the procedure.

parameter-name
Optionally specifies the name of the parameter. The parameter name must be unique for the procedure (SQLSTATE 42734).

data-type
Specifies the data type of the parameter.

- SQL data type specifications and abbreviations, which may be specified in the *data-type* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the procedure, may be specified.

- User-defined data types are not supported (SQLSTATE 42601).

SPECIFIC *specific-name*
Provides a unique name for the instance of the procedure that is being defined. This specific name can be used when dropping the procedure or commenting on the procedure. It can never be used to invoke the procedure. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another routine instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *procedure-name*.

If no qualifier is specified, the qualifier that was used for *procedure-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *procedure-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmsshhn.

DYNAMIC RESULT SETS *integer*
Indicates the estimated upper bound of returned result sets for the stored procedure.

NO SQL, CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA
Indicates whether the stored procedure issues any SQL statements and, if so, what type.

NO SQL
Indicates that the stored procedure cannot execute any SQL statements (SQLSTATE 38001).

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the stored procedure (SQLSTATE 38004). Statements that are not supported in any stored procedure return a different error (SQLSTATE 38003).

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the stored procedure (SQLSTATE 38002 or 42985). Statements that are not supported in any stored procedure return a different error (SQLSTATE 38003).

MODIFIES SQL DATA
Indicates that the stored procedure can execute any SQL statement except statements that are not supported in stored procedures (SQLSTATE 38003).

DETERMINISTIC or **NOT DETERMINISTIC**
This clause specifies whether the procedure always returns the same results for given argument values (DETERMINISTIC) or whether the procedure depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC procedure must always return the same result from successive invocations with identical inputs.

This clause currently does not impact processing of the stored procedure.

CALLED ON NULL INPUT
CALLED ON NULL INPUT always applies to stored procedures. This means that the stored procedure is called regardless of whether any arguments are null. Any OUT or INOUT parameter can return a null value or a normal (non-null) value. Responsibility for testing for null argument values lies with the stored procedure.

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the stored procedure body is written.

C This means the database manager will call the stored procedure as if it were a C procedure. The stored procedure must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA
This means the database manager will call the stored procedure as a method in a Java class.

COBOL
This means the database manager will call the procedure as if it were a COBOL procedure.

OLE
This means the database manager will call the stored procedure as if it were a method exposed by an OLE automation object. The stored-procedure must conform with the OLE automation data types and invocation mechanism. Also, the OLE automation object needs to be implemented as an in-process server (DLL). These restrictions are outlined in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is only supported for stored procedures stored in DB2 for Windows operating systems. THREADSAFE may not be specified for procedures defined with LANGUAGE OLE (SQLSTATE 42613).

EXTERNAL
This clause indicates that the CREATE PROCEDURE statement is being

used to register a new procedure based on code written in an external programming language and adhering to the documented linkage conventions and interface.

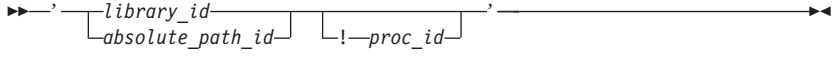
If the NAME clause is not specified, "NAME *procedure-name*" is assumed.

NAME *'string'*
This clause identifies the name of the user-written code which implements the procedure being defined.

The 'string' option is a string constant with a maximum of 254 characters. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

  The *string* specified is the library name and procedure within the library, which the database manager invokes to execute the stored procedure being CREATEd. The library (and the procedure within the library) do not need to exist when the CREATE PROCEDURE statement is performed. However, when the procedure is called, the library and procedure within the library must exist and be accessible from the database server machine.



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

library_id
Identifies the library name containing the procedure. The database manager will look for the library in the .../sqllib/function/unfenced directory and the .../sqllib/function directory (UNIX-based systems), or ...*instance_name*\function\unfenced directory and the ...*instance_name*\function directory (Windows operating systems, as specified by the DB2INSTPROF registry variable), where the database manager will locate the controlling sqllib directory which is being used to run the database manager. For example, the controlling sqllib directory in UNIX-based systems is /u/$DB2INSTANCE/sqllib.

If 'myproc' were the *library_id* in a UNIX-based system it would cause the database manager to look for the procedure in library /u/production/sqllib/function/unfenced/myfunc and /u/production/sqllib/function/myfunc, provided the database manager is being run from /u/production.

For Windows operating systems, the database manager will look in the LIBPATH or PATH if the *library_id* is not found in the function directory.

Stored procedures located in any of these directories do not use any of the registered attributes.

absolute_path_id
Identifies the full path name of the procedure.

In a UNIX-based system, for example, '/u/jchui/mylib/myproc' would cause the database manager to look in /u/jchui/mylib for the myproc procedure.

In Windows operating systems, 'd:\mylib\myproc' would cause the database manager to load the myproc.dll file from the d:\mylib directory.

! proc_id
Identifies the entry point name of the procedure to be invoked. The ! serves as a delimiter between the library id and the procedure id. If *! proc_id* is omitted, the database manager will use the default entry point established when the library was linked.

In a UNIX-based system, for example, 'mymod!proc8' would direct the database manager to look for the library $inst_home_dir/sqllib/function/mymod and to use entry point proc8 within that library.

In Windows operating systems, 'mymod!proc8' would direct the database manager to load the mymod.dll file and call the proc8() procedure in the dynamic link library (DLL).

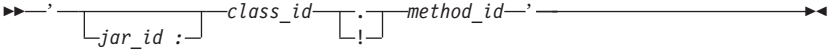
If the string is not properly formed, an error (SQLSTATE 42878) is raised.

The body of every stored procedure should be in a directory that is mounted and available on every partition of the database.

- For LANGUAGE JAVA:

  The *string* specified contains the optional jar file identifier, class identifier and method identifier, which the database manager invokes to execute the stored procedure being CREATEd. The class identifier and method identifier do not need to exist when the CREATE PROCEDURE statement is performed. If a *jar_id* is specified, it must exist when the CREATE PROCEDURE statement is performed. However, when the procedure is called, the class

identifier and the method identifier must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42884) is raised.



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

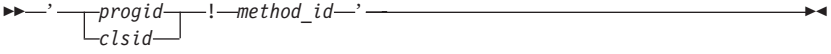
jar_id
Identifies the jar identifier given to the jar collection when it was installed in the database. It can be either a simple identifier or a schema qualified identifier. Examples are 'myJar' and 'mySchema.myJar'.

class_id
Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix, for example, 'myPacks.StoredProcs'. The Java virtual machine will look in directory '../myPacks/StoredProcs/' for the classes. In Windows operating systems, the Java virtual machine will look in directory '..\myPacks\StoredProcs\'.

method_id
Identifies the method name with the Java class to be invoked.

- For LANGUAGE OLE:

  The string specified is the OLE programmatic identifier (*progid*) or class identifier (*clsid*), and method identifier (*method_id*), which the database manager invokes to execute the stored procedure being created by the statement. The programmatic identifier or class identifier, and the method identifier do not need to exist when the CREATE PROCEDURE statement is executed. However, when the procedure is used in the CALL statement, the method identifier must exist and be accessible from the database server machine, otherwise an error results (SQLSTATE 42724).



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

progid
Identifies the programmatic identifier of the OLE object.

A *progid* is not interpreted by the database manager, but only forwarded to the OLE automation controller at run time. The

specified OLE object must be creatable and support late binding (also known as IDispatch-based binding). By convention, progids have the following format:

```
<program_name>.<component_name>.<version>
```

Because this is only a convention, and not a rule, *progids* may in fact have a different format.

clsid
Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a progid in the case that an OLE object is not registered with a progid. The *clsid* has the form:

```
{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}
```

where 'n' is an alphanumeric character. A *clsid* is not interpreted by the database manager, but only forwarded to the OLE APIs at run time.

method_id
Identifies the method name of the OLE object to be invoked.

NAME *identifier*
This *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

FEDERATED or **NOT FEDERATED**
This optional clause specifies whether or not federated objects can be used.

If FEDERATED is specified, federated objects can be accessed from SQL statements in the procedure. Statements within the procedure that access federated objects may be subject to special authorization rules.

If NOT FEDERATED is specified, federated objects cannot be used in any SQL statement in the procedure. Using a federated object will result in an error (SQLSTATE 55047).

FENCED or **NOT FENCED**
This clause specifies whether the stored procedure is considered “safe” to run in the database manager operating environment’s process or address space (NOT FENCED), or not (FENCED).

If a stored procedure is registered as FENCED, the database manager protects its internal resources (for example, data buffers) from access by the procedure. All procedures have the option of running as FENCED or

NOT FENCED. In general, a procedure running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION:
Use of NOT FENCED for procedures that have not been adequately checked out can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that could occur, but cannot guarantee complete integrity when NOT FENCED stored procedures are used.

Either SYSADM authority, DBADM authority, or a special authority (CREATE_NOT_FENCED) is required to register a stored procedure as NOT FENCED. Only FENCED can be specified for a stored procedure with LANGUAGE OLE or NOT THREADSAFE.

THREADSAFE or **NOT THREADSAFE**
Specifies whether the procedure is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the procedure is defined with LANGUAGE other than OLE:

- If the procedure is defined as THREADSAFE, the database manager can invoke the procedure in the same process as other routines. In general, to be threadsafe, a procedure should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED procedures can be THREADSAFE.
- If the procedure is defined as NOT THREADSAFE, the database manager will never invoke the procedure in the same process as another routine.

For FENCED procedures, THREADSAFE is the default if the LANGUAGE is JAVA. For all other languages, NOT THREADSAFE is the default. If the procedure is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).

For NOT FENCED procedures, THREADSAFE is the default. NOT THREADSAFE cannot be specified (SQLSTATE 42613).

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the procedure will inherit their initial values from the environment of the invoking statement.

No changes to the special registers are passed back to the caller of the procedure.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from stored procedures.

DB2GENERAL
This means that the stored procedure will use a parameter passing convention that is defined for use with Java methods. This can only be specified when LANGUAGE JAVA is used.

DB2SQL
In addition to the parameters on the CALL statement, the following arguments are passed to the stored procedure:

- A vector containing a null indicator for each parameter on the CALL statement
- The SQLSTATE to be returned to DB2
- The qualified name of the stored procedure
- The specific name of the stored procedure
- The SQL diagnostic string to be returned to DB2

This can only be specified when LANGUAGE C, COBOL, or OLE is used.

GENERAL
This means that the stored procedure will use a parameter passing mechanism by which the stored procedure receives the parameters specified on the CALL. The parameters are passed directly, as expected by the language; the SQLDA structure is not used. This can only be specified when LANGUAGE C or COBOL is used.

Null indicators are *not* directly passed to the program.

GENERAL WITH NULLS
In addition to the parameters on the CALL statement specified under GENERAL, another argument is passed to the stored procedure. This additional argument is a vector of null indicators, one for each of the parameters on the CALL statement. In C, this would be an array of short ints. This can only be specified when LANGUAGE C or COBOL is used.

JAVA
This means that the stored procedure will use a parameter passing convention that conforms to the Java language and SQLj Routines

specification. IN/OUT and OUT parameters will be passed as single entry arrays to facilitate returning values. This can only be specified when LANGUAGE JAVA is used.

PARAMETER STYLE JAVA procedures do not support the DBINFO or PROGRAM TYPE clauses.

SQL
In addition to the parameters on the CALL statement, the following arguments are passed to the stored procedure:

- A null indicator for each parameter on the CALL statement
- The SQLSTATE to be returned to DB2
- The qualified name of the stored procedure
- The specific name of the stored procedure
- The SQL diagnostic string to be returned to DB2

This can only be specified when LANGUAGE C, COBOL, or OLE is used.

PROGRAM TYPE
Specifies whether the stored procedure expects parameters in the style of a main routine or a subroutine.

SUB
The stored procedure expects the parameters to be passed as separate arguments.

MAIN
The stored procedure expects the parameters to be passed as an argument counter, and a vector of arguments (argc, argv). The name of the stored procedure to be invoked must also be "main". Stored procedures of this type must still be built in the same fashion as a shared library, rather than a stand-alone executable.

The default for PROGRAM TYPE is SUB. PROGRAM TYPE MAIN is only valid for LANGUAGE C or COBOL, and PARAMETER STYLE GENERAL, GENERAL WITH NULLS, SQL, or DB2SQL.

DBINFO or **NO DBINFO**
Specifies whether specific information known by DB2 is passed to the stored procedure when it is invoked as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613). It is also not supported for PARAMETER STYLE JAVA or DB2GENERAL.

If DBINFO is specified, a structure containing the following information is passed to the stored procedure:

- Data base name - the name of the currently connected database.

- Application ID - unique application ID which is established for each connection to the database.
- Application Authorization ID - the application run-time authorization ID.
- Code page - identifies the database code page.
- Database version/release - identifies the version, release and modification level of the database server invoking the stored procedure.
- Platform - contains the server's platform type.

The DBINFO structure is common for all external routines and contains additional fields that are not relevant to procedures.

Notes:

- ***Compatibilities***
  - For compatibility with DB2 UDB for OS/390 and z/OS:
    - The following syntax is accepted as the default behavior:
      - ASUTIME NO LIMIT
      - COMMIT ON RETURN NO
      - NO COLLID
      - STAY RESIDENT NO
  - For compatibility with previous versions of DB2:
    - RESULT SETS can be specified in place of DYNAMIC RESULT SETS.
    - NULL CALL can be specified in place of CALLED ON NULL INPUT.
    - DB2GENRL can be specified in place of DB2GENERAL.
    - SIMPLE CALL can be specified in place of GENERAL.
    - SIMPLE CALL WITH NULLS can be specified in place of GENERAL WITH NULLS.
    - PARAMETER STYLE DB2DARI is supported.
- Creating a procedure with a schema name that does not already exist results in the implicit creation of that schema, provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- ***Privileges***
  - The definer of a procedure always receives the EXECUTE privilege WITH GRANT OPTION on the procedure, as well as the right to drop the procedure.
  - When the procedure is used in an SQL statement, the procedure definer must have the EXECUTE privilege on any packages used by the procedure.

Examples:

Example 1: Create the procedure definition for a stored procedure, written in Java, that is passed a part number and that returns the cost of the part and the quantity that is currently available.

```
CREATE PROCEDURE PARTS_ON_HAND (IN PARTNUM INTEGER,
    OUT COST DECIMAL(7,2),
    OUT QUANTITY INTEGER)
EXTERNAL NAME 'parts.onhand'
LANGUAGE JAVA PARAMETER STYLE JAVA
```

Example 2: Create the procedure definition for a stored 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)
  EXTERNAL NAME 'parts!assembly'
  DYNAMIC RESULT SETS 1 NOT FENCED
  LANGUAGE C PARAMETER STYLE GENERAL
```

Related reference:

- "SQL statements allowed in routines" in the *SQL Reference, Volume 1*
- "Special registers" in the *SQL Reference, Volume 1*

Related samples:

- "spcreate.db2 -- How to catalog the stored procedures contained in spserver.sqc "
- "spcreate.db2 -- Catalog the DB2 CLI stored procedures contained in spserver.sqc "
- "SpCreate.db2 -- How to catalog the stored procedures contained in SpServer.java "
- "SpCreate.db2 -- How to catalog the stored procedures contained in SpServer.sqlj "

CREATE PROCEDURE (SQL)

The CREATE PROCEDURE (SQL) statement is used to define an SQL procedure with an application server.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

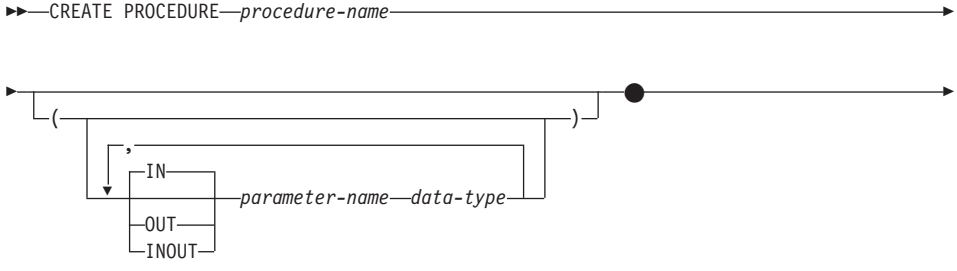
The privileges held by the authorization ID of the statement must include at least one of the following:

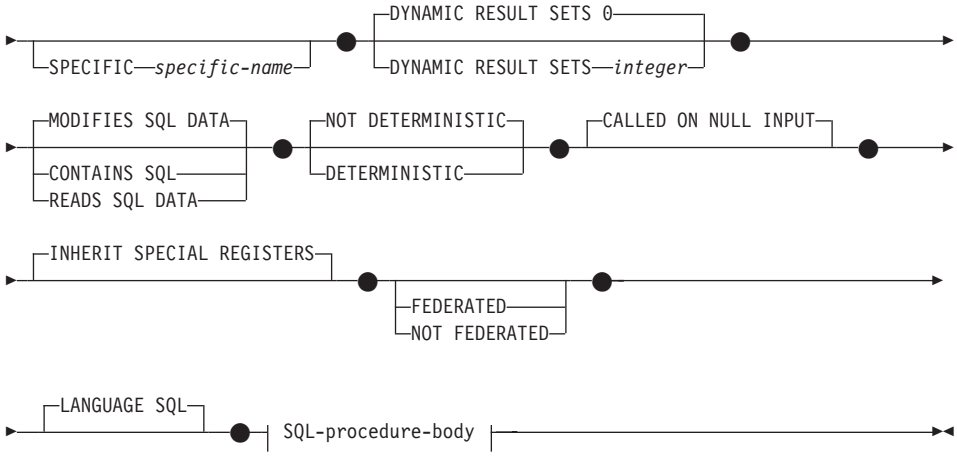
- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the procedure does not exist
- CREATEIN privilege on the schema, if the schema name of the procedure refers to an existing schema.

If the authorization ID of the statement does not have SYSADM or DBADM authority, the privileges that the authorization ID of the statement holds must also include all of the privileges necessary to invoke the SQL statements that are specified in the procedure body.

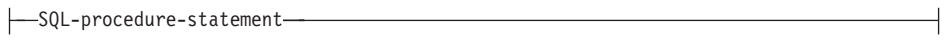
If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

Syntax:





SQL-procedure-body:



Description:

procedure-name
Names the procedure being defined. It is a qualified or unqualified name that designates a procedure. The unqualified form of *procedure-name* is an SQL identifier (with a maximum length of 128). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters, must not identify a procedure described in the catalog (SQLSTATE 42723). The unqualified name, together with the number of parameters, is unique within its schema, but does not need to be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with "SYS". Otherwise, an error (SQLSTATE 42939) is raised.

*(***IN** | **OUT** | **INOUT** *parameter-name data-type,...)*
Identifies the parameters of the procedure, and specifies the mode, name, and data type of each parameter. One entry in the list must be specified for each parameter that the procedure will expect.

It is possible to register a procedure that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example:

```
CREATE PROCEDURE SUBWOOFER() ...
```

No two identically-named procedures within a schema are permitted to have exactly the same number of parameters. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE PROCEDURE PART (IN NUMBER INT, OUT PART_NAME CHAR(35)) ...
CREATE PROCEDURE PART (IN COST DECIMAL(5,3), OUT COUNT INT) ...
```

the second statement will fail because the number of parameters in the procedure is the same, even if the data types are not.

IN | OUT | INOUT
Specifies the mode of the parameter.

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 for the procedure.

INOUT
Identifies the parameter as both an input and output parameter for the procedure.

parameter-name
Specifies the name of the parameter. The parameter name must be unique for the procedure (SQLSTATE 42734).

data-type
Specifies the data type of the parameter.

- SQL data type specifications and abbreviations that can be specified in the *data-type* definition of a CREATE TABLE statement, and that have a correspondence in the language that is being used to write the procedure, may be specified.
- User-defined data types are not supported (SQLSTATE 42601).

SPECIFIC *specific-name*
Provides a unique name for the instance of the procedure that is being defined. This specific name can be used when dropping the procedure or commenting on the procedure. It can never be used to invoke the procedure. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another procedure instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* can be the same as an existing *procedure-name*.

If no qualifier is specified, the qualifier that was used for *procedure-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier for *procedure-name*, or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQL*yymmddhhmmsshhn*.

DYNAMIC RESULT SETS *integer*
Indicates the estimated upper bound of returned result sets for the stored procedure.

CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA
Indicates the level of data access for SQL statements included in the procedure.

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the stored procedure (SQLSTATE 38004 or 42985). Statements that are not supported in any stored procedure return a different error (SQLSTATE 38003 or 42985).

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the stored procedure (SQLSTATE 38002 or 42985). Statements that are not supported in any stored procedure return a different error (SQLSTATE 38003 or 42985).

MODIFIES SQL DATA
Indicates that the stored procedure can execute any SQL statement except statements that are not supported in stored procedures (SQLSTATE 38003 or 42985).

DETERMINISTIC or **NOT DETERMINISTIC**
This clause specifies whether the procedure always returns the same results for given argument values (DETERMINISTIC) or whether the procedure depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC procedure must always return the same result from successive invocations with identical inputs.

This clause currently does not impact processing of the stored procedure.

CALLED ON NULL INPUT
CALLED ON NULL INPUT always applies to stored procedures. This means that the stored procedure is called regardless of whether any arguments are null. Any OUT or INOUT parameter can return a null value or a normal (non-null) value. Responsibility for testing for null argument values lies with the stored procedure.

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the procedure will inherit their initial values from the environment of the invoking statement. For a routine invoked in a nested object (for example a trigger or view), the initial values are inherited from the runtime environment (not inherited from the object definition).

No changes to the special registers are passed back to the caller of the procedure.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

FEDERATED or **NOT FEDERATED**
This optional clause specifies whether or not federated objects can be used.

If neither option is specified, the body of the procedure is examined. If a federated object is referenced in the body of the procedure, a warning is raised (SQLSTATE 01639) and the procedure is created as FEDERATED. If no federated object is referenced, the procedure is defined as NOT FEDERATED.

If FEDERATED is specified, federated objects can be accessed from SQL statements in the procedure. Statements within the procedure that access federated objects may be subject to special authorization rules.

If NOT FEDERATED is specified, federated objects cannot be used in any SQL statement in the procedure. Using a federated object will result in an error (SQLSTATE 429BA).

LANGUAGE SQL
This clause is used to specify that the procedure body is written in the SQL language.

SQL-procedure-body
Specifies the SQL statement that is the body of the SQL procedure. Multiple SQL-procedure-statements can be specified within a compound statement.

Notes:

- ***Compatibilities***
  - For compatibility with DB2 UDB for OS/390 and z/OS:
    - The following syntax is accepted as the default behavior:
      - ASUTIME NO LIMIT
      - COMMIT ON RETURN NO
      - NO COLLID

- STAY RESIDENT NO

– For compatibility with previous versions of DB2:
  - RESULT SETS can be specified in place of DYNAMIC RESULT SETS.
  - NULL CALL can be specified in place of CALLED ON NULL INPUT.
– Creating a procedure with a schema name that does not already exist will result in the implicit creation of that schema, provided that the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
– ***Privileges***

  The definer of a procedure always receives the EXECUTE privilege WITH GRANT OPTION on the procedure, as well as the right to drop the procedure.

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 DOUBLE)
  RESULT SETS 1
  LANGUAGE SQL
BEGIN
  DECLARE v_numRecords INT DEFAULT 1;
  DECLARE v_counter INT DEFAULT 0;

  DECLARE c1 CURSOR FOR
    SELECT CAST(salary AS DOUBLE)
      FROM staff
      ORDER BY salary;
  DECLARE c2 CURSOR WITH RETURN FOR
    SELECT name, job, CAST(salary AS INTEGER)
      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
```

Related reference:

- "SQL statements allowed in routines" in the *SQL Reference, Volume 1*
- "Special registers" in the *SQL Reference, Volume 1*

Related samples:

- "basecase.db2 -- To create the UPDATE_SALARY SQL procedure "
- "nestcase.db2 -- To create the BUMP_SALARY SQL procedure "
- "nestedsp.db2 -- To create the OUT_AVERAGE, OUT_MEADIN and MAX_SALARY SQL procedures "
- "rsultset.db2 -- To create the MEDIAN_RESULT_SET SQL procedure "

CREATE SCHEMA

The CREATE SCHEMA statement defines a schema. It is also possible to create some objects and grant privileges on objects within the statement.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

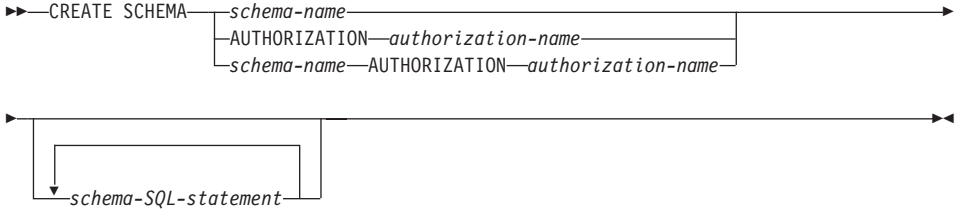
An authorization ID that holds SYSADM or DBADM authority can create a schema with any valid *schema-name* or *authorization-name*.

An authorization ID that does not hold SYSADM or DBADM authority can only create a schema with a *schema-name* or *authorization-name* that matches the authorization ID of the statement.

If the statement includes any *schema-SQL-statement*s the privileges held by the *authorization-name* (if not specified, it defaults to the authorization ID of the statement) must include at least one of the following:

- The privileges required to perform each of the *schema-SQL-statement*s
- SYSADM or DBADM authority.

Syntax:



Description:

schema-name
Names the schema. The name must not identify a schema already described in the catalog (SQLSTATE 42710). The name cannot begin with "SYS" (SQLSTATE 42939). The owner of the schema is the authorization ID that issued the statement.

AUTHORIZATION *authorization-name*
Identifies the user that is the owner of the schema. The value

authorization-name is also used to name the schema. The *authorization-name* must not identify a schema already described in the catalog (SQLSTATE 42710).

schema-name **AUTHORIZATION** *authorization-name*
Identifies a schema called *schema-name* with the user called *authorization-name* as the schema owner. The *schema-name* must not identify a schema-name for a schema already described in the catalog (SQLSTATE 42710). The *schema-name* cannot begin with "SYS" (SQLSTATE 42939).

schema-SQL-statement
SQL statements that can be included as part of the CREATE SCHEMA statement are:

- CREATE TABLE statement, excluding typed tables and materialized query tables
- CREATE VIEW statement, excluding typed views
- CREATE INDEX statement
- COMMENT statement
- GRANT statement

Notes:

- The owner of the schema is determined as follows:
  - If an AUTHORIZATION clause is specified, the specified *authorization-name* is the schema owner
  - If an AUTHORIZATION clause is not specified, the authorization ID that issued the CREATE SCHEMA statement is the schema owner.
- The schema owner is assumed to be a user (not a group).
- When the schema is explicitly created with the CREATE SCHEMA statement, the schema owner is granted CREATEIN, DROPIN, and ALTERIN privileges on the schema with the ability to grant these privileges to other users.
- The definer of any object created as part of the CREATE SCHEMA statement is the schema owner. The schema owner is also the grantor for any privileges granted as part of the CREATE SCHEMA statement.
- Unqualified object names in any SQL statement within the CREATE SCHEMA statement are implicitly qualified by the name of the created schema.
- If the CREATE statement contains a qualified name for the object being created, the schema name specified in the qualified name must be the same as the name of the schema being created (SQLSTATE 42875). Any other objects referenced within the statements may be qualified with any valid schema name.

- If the AUTHORIZATION clause is specified and DCE authentication is used, the group membership of the *authorization-name* specified will not be considered in evaluating the authorizations required to perform the statements that follow the clause. If the *authorization-name* specified is different than the authorization id creating the schema, an authorization failure may result during the execution of the CREATE SCHEMA statement.
- It is recommended not to use ″SESSION″ as a schema name. Since declared temporary tables must be qualified by ″SESSION″, it is possible to have an application declare a temporary table with a name identical to that of a persistent table. An SQL statement that references a table with the schema name ″SESSION″ will resolve (at statement compile time) to the declared temporary table rather than a persistent table with the same name. Since an SQL statement is compiled at different times for static embedded and dynamic embedded SQL statements, the results depend on when the declared temporary table is defined. If persistent tables, views or aliases are not defined with a schema name of ″SESSION″, these issues do not require consideration.

Examples:

Example 1: As a user with DBADM authority, create a schema called RICK with the user RICK as the owner.

```
CREATE SCHEMA RICK AUTHORIZATION RICK
```

Example 2: Create a schema that has an inventory part table and an index over the part number. Give authority on the table to user JONES.

```
CREATE SCHEMA INVENTRY

  CREATE TABLE PART (PARTNO   SMALLINT NOT NULL,
                     DESCR    VARCHAR(24),
                     QUANTITY INTEGER)

  CREATE INDEX PARTIND ON PART (PARTNO)

  GRANT ALL ON PART TO JONES
```

Example 3: Create a schema called PERS with two tables that each have a foreign key that references the other table. This is an example of a feature of the CREATE SCHEMA statement that allows such a pair of tables to be created without the use of the ALTER TABLE statement.

```
CREATE SCHEMA PERS

  CREATE TABLE ORG (DEPTNUMB  SMALLINT NOT NULL,
                    DEPTNAME VARCHAR(14),
                    MANAGER  SMALLINT,
                    DIVISION VARCHAR(10),
                    LOCATION VARCHAR(13),
                    CONSTRAINT PKEYDNO
```

```
                        PRIMARY KEY (DEPTNUMB),
                      CONSTRAINT FKEYMGR
                        FOREIGN KEY (MANAGER)
                        REFERENCES STAFF (ID) )

CREATE TABLE STAFF (ID        SMALLINT NOT NULL,
                    NAME      VARCHAR(9),
                    DEPT      SMALLINT,
                    JOB       VARCHAR(5),
                    YEARS     SMALLINT,
                    SALARY    DECIMAL(7,2),
                    COMM      DECIMAL(7,2),
                    CONSTRAINT PKEYID
                      PRIMARY KEY (ID),
                    CONSTRAINT FKEYDNO
                      FOREIGN KEY (DEPT)
                      REFERENCES ORG (DEPTNUMB) )
```

Related reference:

- "COMMENT" on page 109
- "CREATE INDEX" on page 268
- "CREATE TABLE" on page 332
- "CREATE VIEW" on page 463
- "GRANT (Table, View, or Nickname Privileges)" on page 590

CREATE SEQUENCE

The CREATE SEQUENCE statement creates a sequence at the application server.

Invocation:

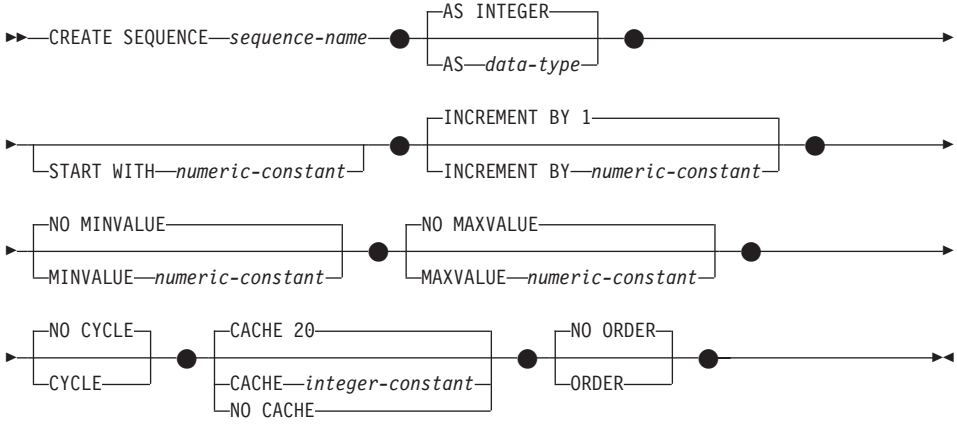
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- CREATEIN privilege for the implicitly or explicitly specified schema
- SYSADM or DBADM authority

Syntax:



Description:

sequence-name

Names the sequence. The combination of name, and the implicit or explicit schema name must not identify an existing sequence at the current server (SQLSTATE 42710).

The unqualified form of sequence-name is an SQL identifier. The qualified form is a qualifier followed by a period and an SQL identifier. The qualifier is a schema name.

If the sequence name is explicitly qualified with a schema name, the schema name cannot begin with 'SYS' or an error (SQLSTATE 42939) is raised.

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 or reference type for which the source type is an exact numeric type with a scale of zero (SQLSTATE 42815). The default is INTEGER.

START WITH *numeric-constant*
Specifies the first value for 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 (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA). 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. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815), and does not exceed the value of a large integer constant (SQLSTATE 42820), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA).

If this value is negative, this is a descending sequence. If this value is 0 or positive, this is an ascending sequence. The default is 1.

MINVALUE or NO 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.

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 (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be less than or equal to the maximum value (SQLSTATE 42815).

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. This is the default.

MAXVALUE or NO 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.

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 (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be greater than or equal to the minimum value (SQLSTATE 42815).

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.

CYCLE or NO CYCLE
Specifies whether the sequence should continue to generate values after reaching either its maximum or minimum value. The boundary of the sequence can be reached either with the next value landing exactly on the boundary condition, or by overshooting it.

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; 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.

NO CYCLE
Specifies that values will not be generated for the sequence once the maximum or minimum value for the sequence has been reached. This is the default.

CACHE or NO CACHE
Specifies whether to keep some preallocated values in memory for faster access. This is a performance and tuning option.

CACHE *integer-constant*
Specifies the maximum number of sequence values that are preallocated and kept in memory. Preallocating and storing values in the cache reduces synchronous I/O to the log when values are generated for the sequence.

In the event of a system failure, all cached sequence values that have not been used in committed statements are lost (that is, they will never be used). The value specified for the CACHE option is the maximum number of sequence values that could be lost in case of system failure.

The minimum value is 2 (SQLSTATE 42815). The default value is CACHE 20.

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 synchronous I/O to the log.

NO ORDER or ORDER
Specifies whether the sequence numbers must be generated in order of request.

ORDER
Specifies that the sequence numbers are generated in order of request.

NO ORDER
Specifies that the sequence numbers do not need to be generated in order of request. This is the default.

Notes:

- Sequences are not supported in a database with multiple partitions (SQLSTATE 42997). A sequence cannot be created if more than one partition for the database exists. A database that includes any sequence objects cannot be started with more than one partition.
- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - A comma can be used to separate multiple sequence options
  - The following syntax is also supported:

  - NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER.

- It is possible to define a constant sequence, that is, one 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 NEXTVAL is invoked for the sequence, 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.
- 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 (i.e., CYCLE was explicitly specified), the maximum or minimum value generated for a sequence may not be the actual MAXVALUE or MINVALUE specified if the increment is a value other than 1 or -1. For example, the sequence defined with `START WITH=1, INCREMENT=2, MAXVALUE=10` will generate a maximum value of 9, and will not generate the value 10. When defining a sequence with CYCLE, carefully consider the impact of the values for MINVALUE, MAXVALUE and START WITH.
- 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 for I/O operations to persistent storage. The choice of the value for CACHE should be done keeping in mind the performance and application requirements tradeoffs.
- The owner of a sequence has ALTER and USAGE privileges on the sequence. Only the USAGE privilege can be granted by the owner, and only to PUBLIC. The definer of a sequences is granted ALTER and USAGE privileges with the grant option. The definer can also drop the sequence.

Examples:

Example 1: Create a sequence called ORG_SEQ that starts at 1, increments by 1, does not cycle, and caches 24 values at a time:

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

Related samples:

- "DbSeq.java -- How to create, alter and drop a sequence in a database (JDBC)"

CREATE SERVER

The CREATE SERVER statement defines a data source to a federated database. (In this statement, the term SERVER and the parameter names that start with *server-* refer only to data sources in a federated system. They do not refer to the federated server in such a system, or to DRDA application servers.)

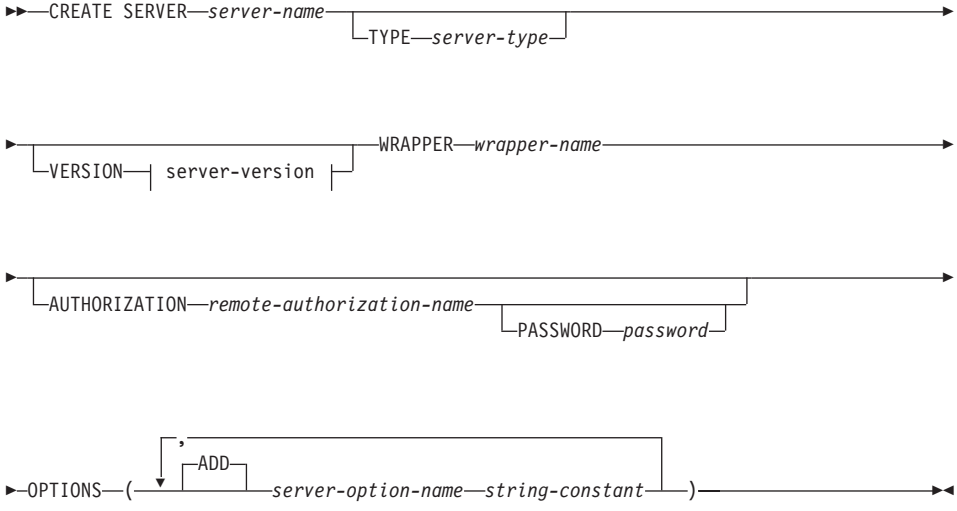
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

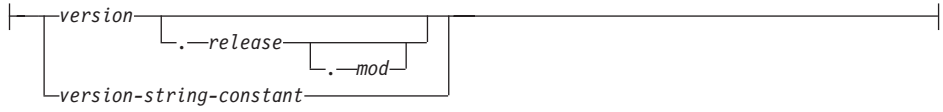
Authorization:

The authorization ID of the statement must have SYSADM or DBADM authority on the federated database.

Syntax:



server-version:



Description:

server-name
Names the data source that is being defined to the federated database. The name must not identify a data source that is described in the catalog. The *server-name* must not be the same as the name of any table space in the federated database.

TYPE *server-type*
Specifies the type of data source denoted by *server-name*. This option is required for all relational wrappers: DRDA, SQLNET, NET8, INFORMIX, CTLIB, DBLIB, MSSQLODBC3, and DJXMSSQL3.

VERSION
Specifies the version of the data source denoted by *server-name.*

version
Specifies the version number. *version* must be an integer.

release
Specifies the number of the release of the version denoted by *version. release* must be an integer.

mod
Specifies the number of the modification of the release denoted by *release. mod* must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release*, and, if applicable, *mod* (for example, '8.0.3').

WRAPPER *wrapper-name*
Names the wrapper that the federated server uses to interact with data sources of the type and version denoted by *server-type* and *'server-version'*.

AUTHORIZATION *remote-authorization-name*
Specifies the authorization ID under which any necessary actions are performed at the data source when the CREATE SERVER statement is processed. This ID must hold the authority (BINDADD or its equivalent) that the necessary actions require. If the *remote-authorization-name* is specified in mixed or lowercase characters (and the remote data source has case sensitive authorization names), it should be enclosed by double quotation marks.

PASSWORD *password*
Specifies the password associated with the authorization ID represented by *remote-authorization-name.* If *password* is not specified, it will default to the password for the ID under which the user is connected to the federated database. If the *password* is specified in mixed or lowercase

characters (and the remote data source has case sensitive passwords), it should be enclosed by double quotation marks.

OPTIONS
Indicates what server options are to be enabled.

ADD
Enables one or more server options.

server-option-name
Names a server option that will be used to either configure or provide information about the data source denoted by *server-name*.

string-constant
Specifies the setting for *server-option-name* as a character string constant.

Notes:

- If *remote-authorization-name* is not specified, the authorization ID for the federated database will be used.
- The *password* should be specified in the case required by the data source; if any letters in *password* must be in lowercase, enclose *password* in quotation marks. If an *identifier* is specified but not *password*, the authentication type of the data source denoted by *server-name* is assumed to be CLIENT.
- If the CREATE SERVER statement is used to define a DB2 family instance as a data source, DB2 may need to bind certain packages to that instance. If a bind is required, the *remote-authorization-name* in the statement must have BIND authority. The time required for the bind to complete is dependent on data source speed and network connection speed.
- If a server option is set to one value for a type of data source, and this same option set to another value for an instance of this type, the second value overrides the first for the instance. For example, suppose that PASSWORD is set to 'Y' (yes, validate passwords at the data source) for a federated system's DB2 Universal Database for OS/390 and z/OS data sources. Then later, this option's default ('N') is used for a specific DB2 Universal Database for OS/390 and z/OS data source named SIBYL. As a result, passwords will be validated at all of the DB2 Universal Database for OS/390 and z/OS data sources except SIBYL.

Examples:

Example 1: Define a DB2 for MVS/ESA 4.1 data source that is accessible through a wrapper called DB2WRAP. Call the data source CRANDALL. In addition, specify that:

- MURROW and DROWSSAP will be the authorization ID and password under which packages are bound at CRANDALL when this statement is processed.

- CRANDALL is defined to the DB2 RDBMS as an instance called MYNODE.
- When the federated server accesses CRANDALL, it will be connected to a database called MYDB.
- The authorization IDs and passwords under which CRANDALL can be accessed are to be sent to CRANDALL in uppercase.
- MYDB and the federated database use the same collating sequence.

```
CREATE SERVER CRANDALL
  TYPE DB2/MVS
  VERSION 4.1
  WRAPPER DB2WRAP
  AUTHORIZATION MURROW
  PASSWORD DROWSSAP
  OPTIONS
      ( NODE 'MYNODE',
      DBNAME 'MYDB',
      FOLD_ID 'U',
      FOLD_PW 'U',
      COLLATING_SEQUENCE 'Y' )
```

Example 2: Define an Oracle 7.2 data source that's accessible through a wrapper called KLONDIKE. Call the data source CUSTOMERS. Specify that:

- CUSTOMERS is defined to the Oracle RDBMS as an instance called ABC.

Provide these statistics for the optimizer:

- The CPU for the federated server runs twice as fast as the CPU that supports CUSTOMERS.
- The I/O devices at the federated server process data one and a half times as fast as the I/O devices at CUSTOMERS.

```
CREATE SERVER CUSTOMERS
  TYPE ORACLE
  VERSION 7.2
  WRAPPER KLONDIKE
  OPTIONS
      ( NODE 'ABC',
      CPU_RATIO '2.0',
      IO_RATIO '1.5' )
```

Related concepts:

- "Distributed relational databases" in the *SQL Reference, Volume 1*

Related reference:

- "Server options for federated systems" in the *Federated Systems Guide*

CREATE TABLE

The CREATE TABLE statement defines a table. 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 or check constraints.

To declare a global temporary table, use the DECLARE GLOBAL TEMPORARY TABLE statement.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATETAB authority on the database and USE privilege on the table space as well as one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the table does not exist
  - CREATEIN privilege on the schema, if the schema name of the table refers to an existing schema.

  If a subtable is being defined, the authorization ID must be the same as the definer of the root table of the table hierarchy.

To define a foreign key, the privileges held by the authorization ID of the statement must include one of the following on the parent table:

- REFERENCES privilege on the table
- REFERENCES privilege on each column of the specified parent key
- CONTROL privilege on the table
- SYSADM or DBADM authority.

To define a materialized query table (using a fullselect) the privileges held by the authorization ID of the statement must include at least one of the following on each table or view identified in the fullselect:

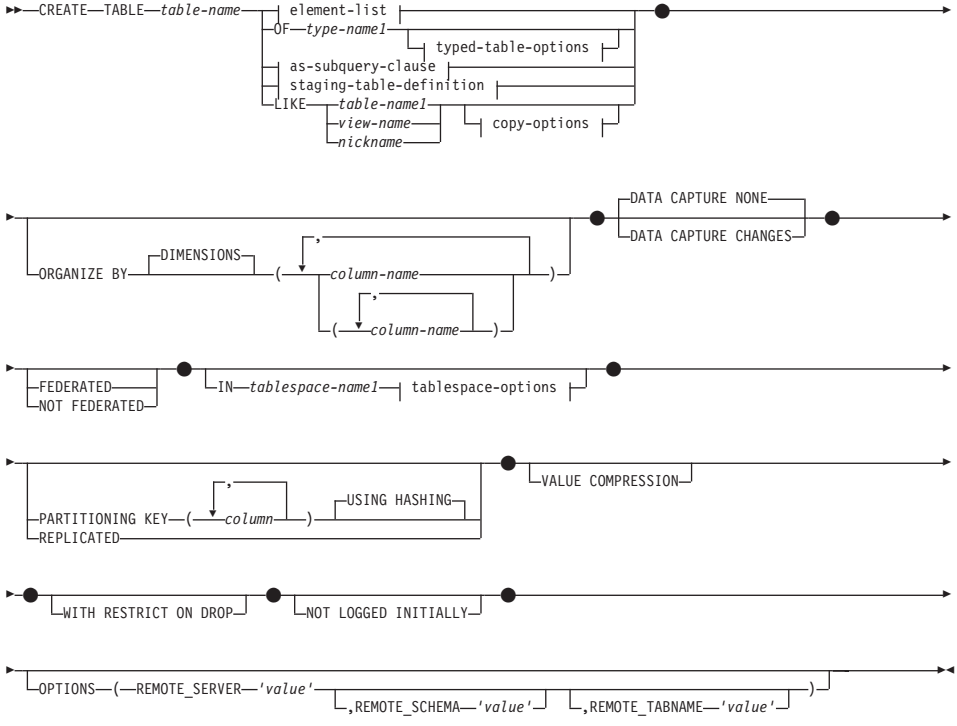
- SELECT privilege on the table or view and ALTER privilege if REFRESH DEFERRED or REFRESH IMMEDIATE is specified

- CONTROL privilege on the table or view
- SYSADM or DBADM authority.

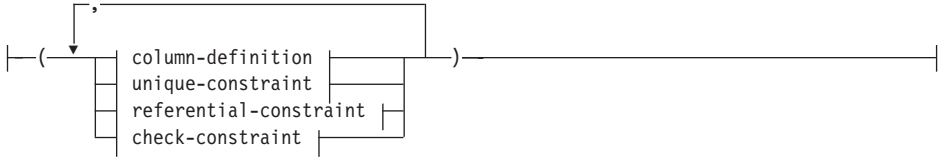
To define a staging table associated with a materialized query table, the privileges held by the authorization ID of the statement must include at least one of the following:

- CONTROL privilege or ALTER privilege on the materialized query table, and at least one of the following on each table or view identified in the fullselect of the materialized query table:
  - SELECT privilege and ALTER privilege on the table or view
  - CONTROL privilege on the table or view
- SYSADM or DBADM authority

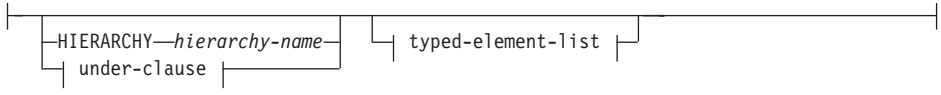
Syntax:



element-list:

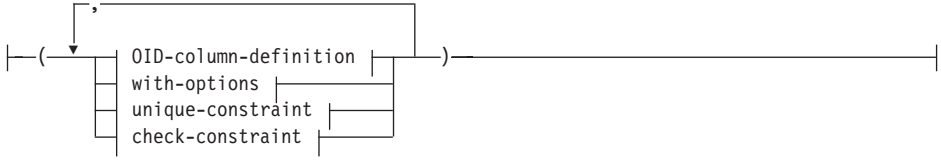


typed-table-options:

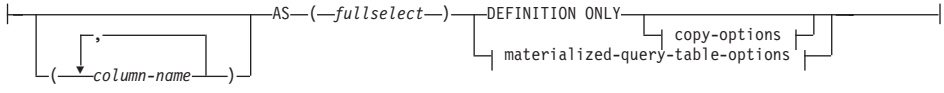


under-clause:

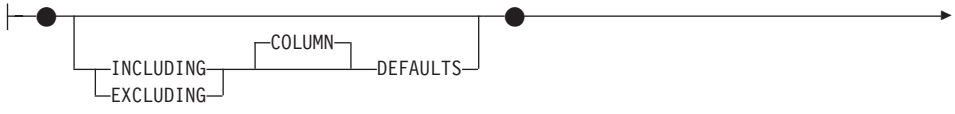
typed-element-list:

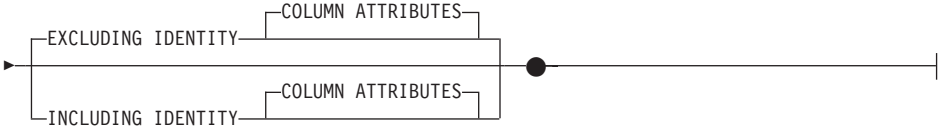


as-subquery-clause:

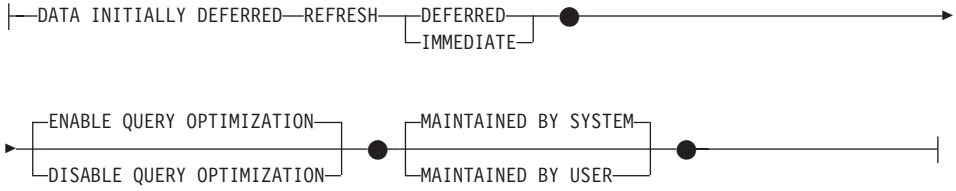


copy-options:

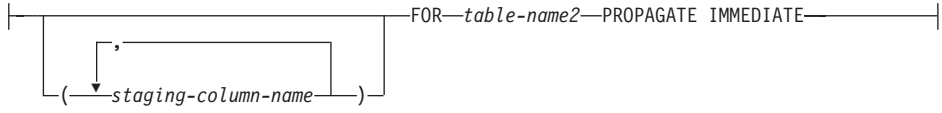




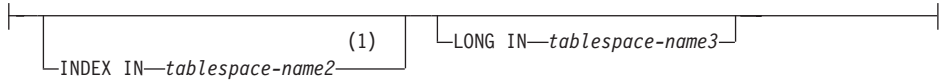
materialized-query-table-options:



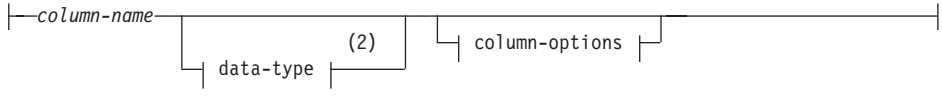
staging-table-definition:



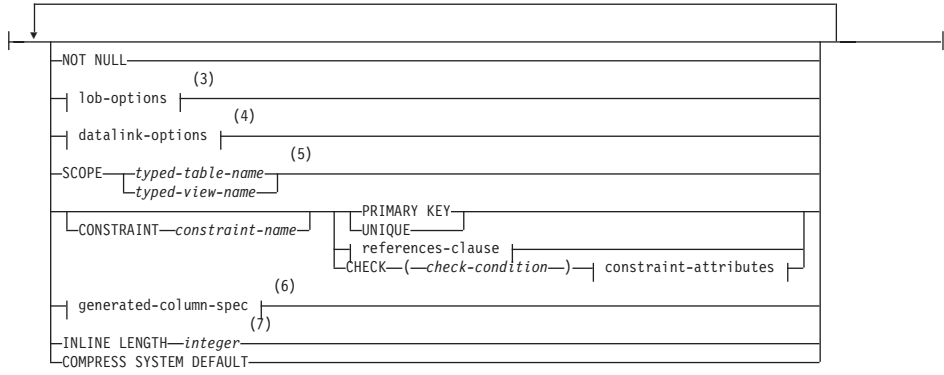
tablespace-options:



column-definition:



column-options:



Notes:

1 Specifying which table space will contain a table's index can only be done when the table is created.

2 If the first column-option chosen is a generated-column-spec with a generation-expression, then the data-type can be omitted. It will be determined from the resulting data type of the generation-expression.

3 The lob-options clause only applies to large object types (BLOB, CLOB and DBCLOB) and distinct types based on large object types.

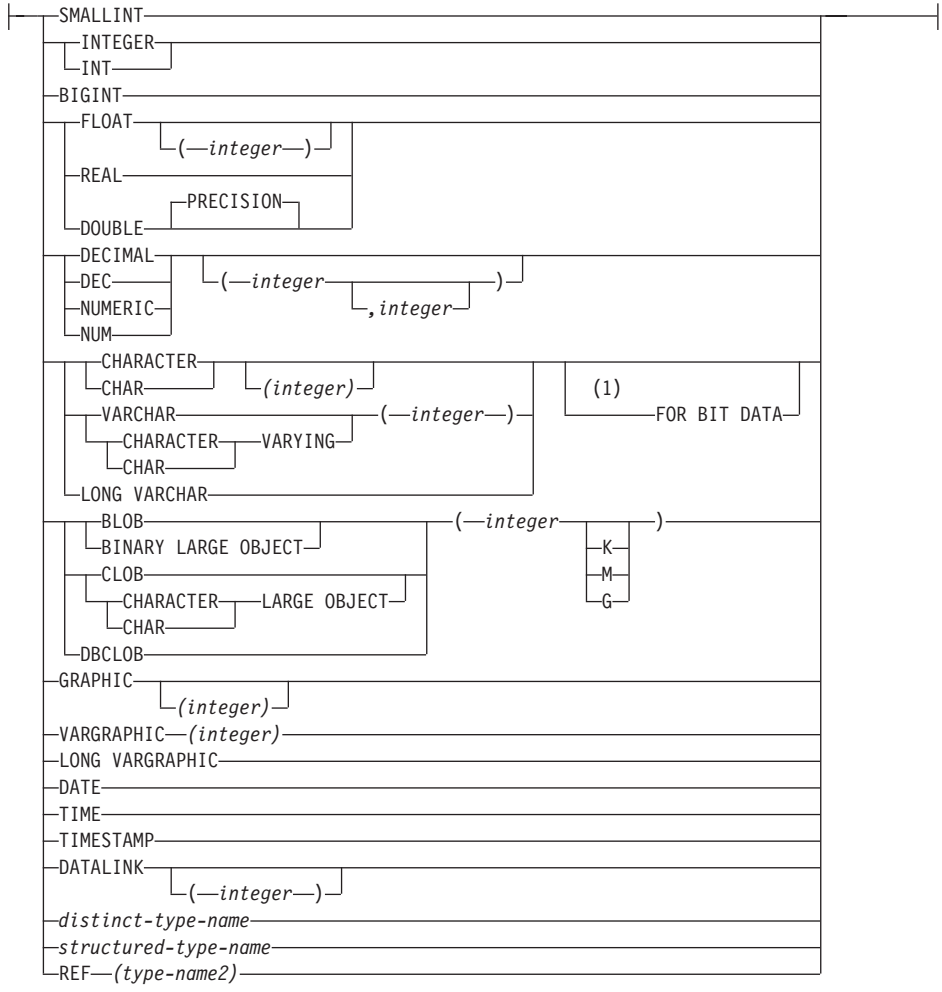
4 The datalink-options clause only applies to the DATALINK type and distinct types based on the DATALINK type.

5 The SCOPE clause only applies to the REF type.

6 IDENTITY column attributes are not supported in a database with more than one partition.

7 INLINE LENGTH only applies to columns defined as structured types.

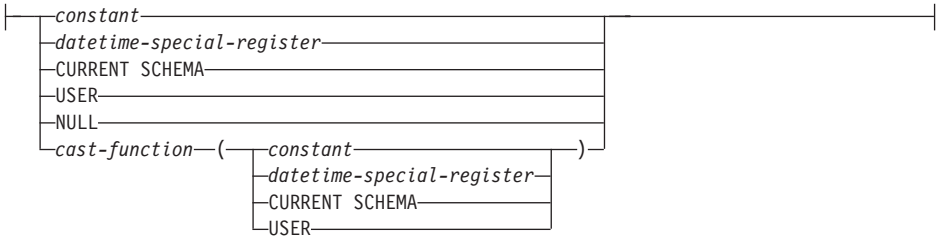
data-type:



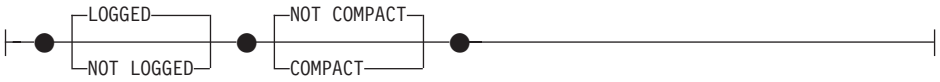
Notes:

1 The FOR BIT DATA clause may be specified in random order with the other column constraints that follow.

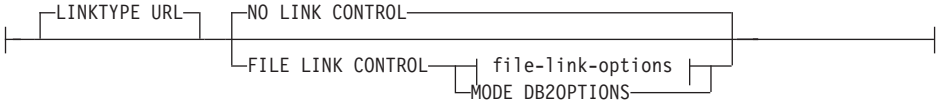
default-values:



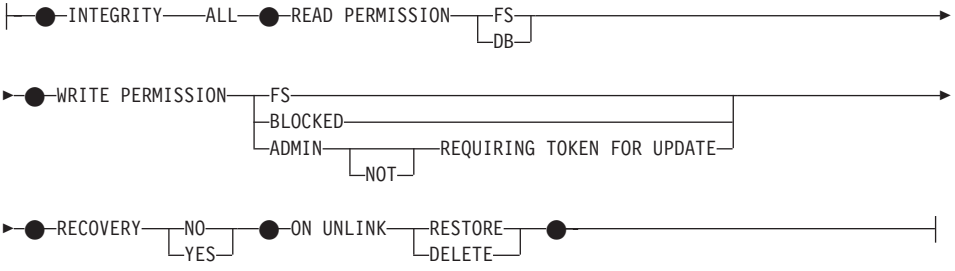
lob-options:



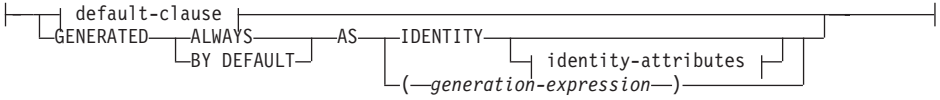
datalink-options:



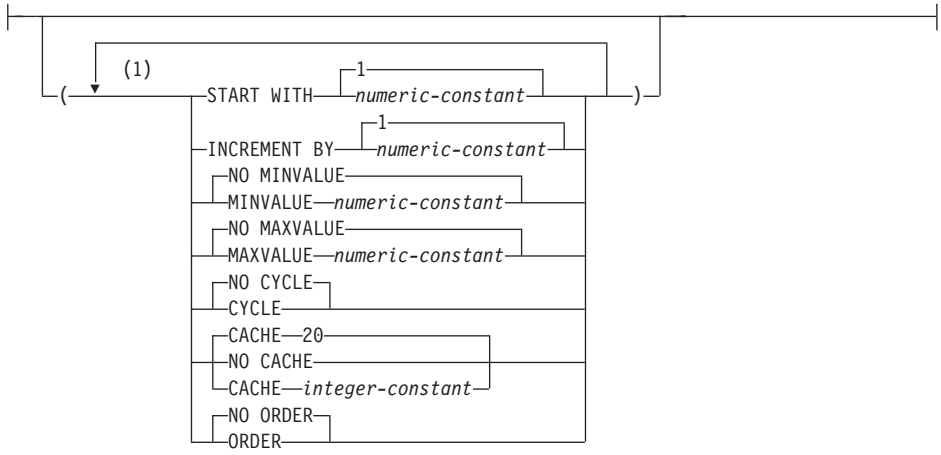
file-link-options:



generated-column-spec:



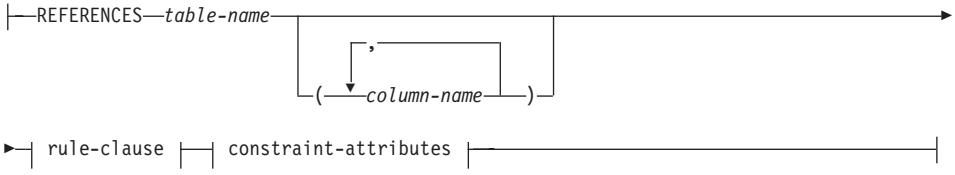
identity-attributes:



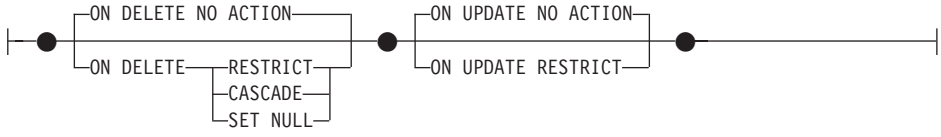
Notes:

1 The same clause must not be specified more than once.

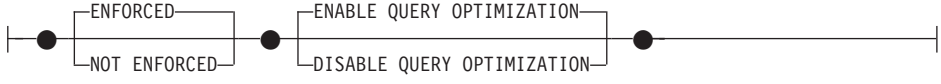
references-clause:



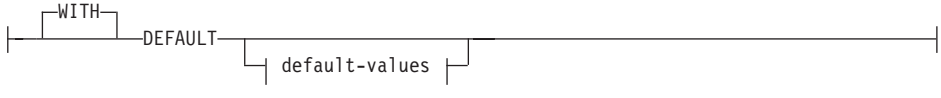
rule-clause:



constraint-attributes:

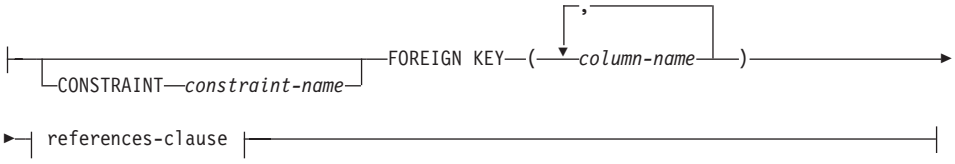


default-clause:

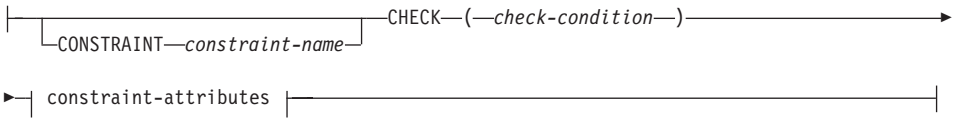


unique-constraint:

referential-constraint:

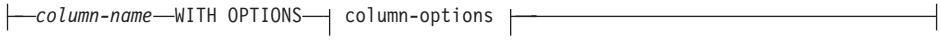


check-constraint:



OID-column-definition:

with-options:



Description:
System-maintained materialized query tables and user-maintained materialized query tables are referred to by the common term *materialized query table*, unless there is a need to identify each one separately.

table-name
Names the table. The name, including the implicit or explicit qualifier, must not identify a table, view, or alias described in the catalog. The schema name must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

OF *type-name1*
Specifies that the columns of the table are based on the attributes of the structured type identified by *type-name1*. If *type-name1* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static

SQL and by the CURRENT PATH register for dynamic SQL). The type name must be the name of an existing user-defined type (SQLSTATE 42704) and it must be an instantiable structured type (SQLSTATE 428DP) with at least one attribute (SQLSTATE 42997).

If UNDER is not specified, an object identifier column must be specified (refer to the *OID-column-definition*). This object identifier column is the first column of the table. The object ID column is followed by columns based on the attributes of *type-name1*.

HIERARCHY *hierarchy-name*
Names the hierarchy table associated with the table hierarchy. It is created at the same time as the root table of the hierarchy. The data for all subtables in the typed table hierarchy is stored in the hierarchy table. A hierarchy table cannot be directly referenced in SQL statements. A *hierarchy-name* is a *table-name*. The *hierarchy-name*, including the implicit or explicit schema name, must not identify a table, nickname, view, or alias described in the catalog. If the schema name is specified, it must be the same as the schema name of the table being created (SQLSTATE 428DQ). If this clause is omitted when defining the root table, a name is generated by the system consisting of the name of the table being created followed by a unique suffix such that the identifier is unique within the identifiers of the existing tables, views, aliases, and nicknames.

UNDER *supertable-name*
Indicates that the table is a subtable of *supertable-name*. The supertable must be an existing table (SQLSTATE 42704) and the table must be defined using a structured type that is the immediate supertype of *type-name1* (SQLSTATE 428DB). The schema name of *table-name* and *supertable-name* must be the same (SQLSTATE 428DQ). The table identified by *supertable-name* must not have any existing subtable already defined using *type-name1* (SQLSTATE 42742).

The columns of the table include the object identifier column of the supertable with its type modified to be REF(*type-name1*), followed by columns based on the attributes of *type-name1* (remember that the type includes the attributes of its supertype). The attribute names cannot be the same as the OID column name (SQLSTATE 42711).

Other table options including table space, data capture, not logged initially and partitioning key options cannot be specified. These options are inherited from the supertable (SQLSTATE 42613).

INHERIT SELECT PRIVILEGES
Any user or group holding a SELECT privilege on the supertable will be granted an equivalent privilege on the newly created subtable. The subtable definer is considered to be the grantor of this privilege.

element-list
Defines the elements of a table. This includes the definition of columns and constraints on the table.

typed-element-list
Defines the additional elements of a typed table. This includes the additional options for the columns, the addition of an object identifier column (root table only), and constraints on the table.

as-subquery-clause
If the table definition is based on the result of a query, the table is a materialized query table based on the query.

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 of an unnamed column (SQLSTATE 42908). An unnamed column is a column derived from a constant, function, expression, or set operation that is not named using the AS clause of the select list.

AS
Introduces the query that is used for the definition of the table and to determine the data included in the table.

fullselect
Defines the query in which the table is based. The resulting column definitions are the same as those for a view defined with the same query.

Every select list element must have a name (use the AS clause for expressions). The *as-subquery-clause* defines attributes of the materialized query table. The option chosen also defines the contents of the fullselect as follows.

When DEFINITION ONLY is specified, any valid fullselect that does not reference a typed table or typed view can be specified.

When REFRESH DEFERRED or REFRESH IMMEDIATE is specified, the fullselect cannot include (SQLSTATE 428EC):

- references to a materialized query table, declared temporary table, or typed table in any FROM clause

- references to a view where the fullselect of the view violates any of the listed restrictions on the fullselect of the materialized query table
- expressions that are a reference type or DATALINK type (or distinct type based on these types)
- functions that have external action
- functions written in SQL
- functions that depend on physical characteristics (for example, DBPARTITIONNUM, HASHEDVALUE)
- table or view references to system objects (Explain tables also should not be specified)
- expressions that are a structured type or LOB type (or a distinct type based on a LOB type)
- user-defined functions defined with either CONTAINS SQL or READS SQL DATA

When REFRESH IMMEDIATE is specified:

- the fullselect must be a subselect
- the fullselect cannot include a reference to a nickname (SQLSTATE 428EC)
- the subselect cannot include:
  - functions that are not deterministic
  - scalar fullselects
  - predicates with fullselects
  - special registers
- a GROUP BY clause must be included in the subselect unless REPLICATED is specified, in which case a GROUP BY clause is not allowed.
- The supported column functions are SUM, COUNT, COUNT_BIG and GROUPING (without DISTINCT). The select list must contain a COUNT(*) or COUNT_BIG(*) column. If the materialized query table select list contains SUM(X), where X is a nullable argument, the materialized query table must also have COUNT(X) in its select list. These column functions cannot be part of any expressions.
- if the FROM clause references more than one table or view, it can only define an inner join without using the explicit INNER JOIN syntax
- all GROUP BY items must be included in the select list

- GROUPING SETS, CUBE and ROLLUP are supported. The GROUP BY items and associated GROUPING column functions in the select list must form a unique key of the result set. Thus, the following restrictions must be satisfied:
  - no grouping sets may be repeated. For example, `ROLLUP(X,Y), X` is not allowed because it is equivalent to `GROUPING SETS((X,Y),(X),(X))`
  - if X is a nullable GROUP BY item that appears within GROUPING SETS, CUBE, or ROLLUP, then GROUPING(X) must appear in the select list
  - grouping on constants is not allowed
- a HAVING clause is not allowed
- if in a multiple partition database partition group, then either the partitioning key must be a subset of the GROUP BY items, or REPLICATED must be specified
- if REPLICATED is specified, the table must have a unique key

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.

DEFINITION ONLY
The query is used only to define the table. The table is not populated using the results of query and the REFRESH TABLE statement cannot be used. When the CREATE TABLE statement is completed, the table is no longer considered a materialized query table.

The columns of the table are defined based on the definitions of the columns that result from the fullselect. If the fullselect references a single table in the FROM clause, select list items that are columns of that table are defined using the column name, data type, and nullability characteristic of the referenced table.

materialized-query-table-options
Define the refreshable options of the materialized query table attributes.

DATA INITIALLY DEFERRED
Data is not inserted into the table as part of the CREATE TABLE statement. A REFRESH TABLE statement specifying the *table-name* is used to insert data into the table.

REFRESH
Indicates how the data in the table is maintained.

DEFERRED
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 the REFRESH TABLE statement is processed. System-maintained materialized query tables defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807). User-maintained materialized query tables defined with this attribute do allow INSERT, UPDATE, or DELETE statements.

IMMEDIATE
The changes made to the underlying tables as part of a DELETE, INSERT, or UPDATE are cascaded to the materialized query table. In this case, the content of the table, at any point-in-time, is the same as if the specified *subselect* is processed. Materialized query tables defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807).

ENABLE QUERY OPTIMIZATION
The materialized query table can be used for query optimization under appropriate circumstances.

DISABLE QUERY OPTIMIZATION
The materialized query table will not be used for query optimization. The table can still be queried directly.

MAINTAINED BY SYSTEM
Indicates that the data in the materialized query table is maintained by the system. This is the default.

MAINTAINED BY USER
Indicates that the data in the materialized query table is maintained by the user. The user is allowed to perform update, delete, or insert operations against user-maintained materialized query tables. The REFRESH TABLE statement, used for system-maintained materialized query tables, cannot be invoked against user-maintained materialized query tables. Only a REFRESH DEFERRED materialized query table can be defined as MAINTAINED BY USER.

staging-table-definition
Defines the query supported by the staging table indirectly through an associated materialized query table. The underlying tables of the materialized query table are also the underlying tables for its associated staging table. The staging table collects changes that need to be applied to the materialized query table to synchronize it with the contents of the underlying tables.

staging-column-name
Names the columns in the staging table. If a list of column names is specified, it must consist of *two* more names than there are columns in the materialized query table for which the staging table is defined. If

the materialized query table is a replicated materialized query table, or the query defining the materialized query table does not contain a GROUP BY clause, the list of column names must consist of *three* more names than there are columns in the materialized query table for which the staging table is defined. 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 associated materialized query table. The additional columns are named GLOBALTRANSID and GLOBALTRANSTIME, and if a third column is necessary, it is named OPERATIONTYPE.

Table 4. Extra Columns Appended in Staging Tables

Column Name	Data Type	Column Description
GLOBALTRANSID	CHAR(8) FOR BIT DATA	The global transaction ID for each propagated row
GLOBALTRANSTIME	CHAR(13) FOR BIT DATA	The timestamp of the transaction
OPERATIONTYPE	SMALLINT	Operation for the propagated row, either insert, update, or delete.

A list of column names must be specified if any of the columns of the associated materialized query table duplicates any of the generated column names (SQLSTATE 42711).

FOR *table-name2*
Specifies the materialized query table that is used for the definition of the staging table. The name, including the implicit or explicit schema, must identify a materialized query table that exists at the current server defined with REFRESH DEFERRED. The fullselect of the associated materialized query table must follow the same restrictions and rules as a fullselect used to create a materialized query table with the REFRESH IMMEDIATE option.

The contents of the staging table can be used to refresh the materialized query table, by invoking the REFRESH TABLE statement, if the contents of the staging table are consistent with the associated materialized query table and the underlying source tables.

PROPAGATE IMMEDIATE
The changes made to the underlying tables as part of a delete, insert, or update operation are cascaded to the staging table in the same delete, insert, or update operation. If the staging table is not marked inconsistent, its content, at any point-in-time, is the delta changes to the underlying table since the last refresh materialized query table.

LIKE *table-name1* **or** *view-name* **or** *nickname*
Specifies that the columns of the table have exactly the same name and description as the columns of the identified table (*table-name1*), view (*view-name*) or nickname (*nickname*). The name specified after LIKE must identify a table, view or nickname that exists in the catalog, or a declared temporary table. A typed table or typed view cannot be specified (SQLSTATE 428EC).

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table, view or nickname.

- If a table is identified, then the implicit definition includes the column name, data type and nullability characteristic of each of the columns of *table-name1*. If EXCLUDING COLUMN DEFAULTS is not specified, then the column default is also included.
- If a view is identified, then the implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of the fullselect defined in *view-name*.
- If a nickname is identified, then the implicit definition includes the column name, data type, and nullability characteristic of each column of *nickname*.

Column default and identity column attributes may be included or excluded, based on the copy-attributes clauses. The implicit definition does not include any other attributes of the identified table, view or nickname. Thus the new table does not have any unique constraints, foreign key constraints, triggers, or indexes. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

copy-options
These options specify whether or not to copy additional attributes of the source result table definition (table, view or fullselect).

INCLUDING COLUMN DEFAULTS
Column defaults for each updatable column of the source result table definition are copied. Columns that are not updatable will not have a default defined in the corresponding column of the created table.

If LIKE *table-name* is specified and *table-name* identifies a base table or declared temporary table, then INCLUDING COLUMN DEFAULTS is the default.

EXCLUDING COLUMN DEFAULTS
Columns defaults are not copied from the source result table definition.

This clause is the default, except when LIKE *table-name* is specified and *table-name* identifies a base table or declared temporary table.

INCLUDING IDENTITY COLUMN ATTRIBUTES
Identity column attributes are copied from the source result table definition, if possible. It is possible to copy the identity column attributes, if the element of the corresponding column in the table, view, or fullselect is the name of a table column, or the name of a view column which directly or indirectly maps to the name of a base table column with the identity property. In all other cases, the columns of the new table will not get the identity property. For example:

- the select-list of the fullselect includes multiple instances of an identity column name (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, except, or intersect).

EXCLUDING IDENTITY COLUMN ATTRIBUTES
Identity column attributes are not copied from the source result table definition.

ORGANIZE BY DIMENSIONS *(column-name,...)*
Specifies a dimension for each column or group of columns used to cluster the table data. The use of parentheses within the dimension list specifies that a group of columns is to be treated as one dimension. The DIMENSIONS keyword is optional.

A clustering block index is automatically maintained for each specified dimension, and a block index, consisting of all columns used in the clause, is maintained if none of the clustering block indexes includes them all. The set of columns used in the ORGANIZE BY clause must follow the rules for the CREATE INDEX statement.

Each column name specified in the ORGANIZE BY clause must be defined for the table (SQLSTATE 42703), and a dimension cannot occur more than once in the dimension list (SQLSTATE 42709).

Pages of the table are arranged in blocks of equal size, which is the extent size of the tablespace, and all rows of each block contain the same combination of dimension values.

column-definition
Defines the attributes of a column.

column-name
Names a column of the table. The name cannot be qualified, and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

A table may have the following:

- A 4K page size with a maximum of 500 columns, where the byte counts of the columns must not be greater than 4 005.
- An 8K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 8 101.
- A 16K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 16 293.
- A 32K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 32 677.

For more details, see "Row Size" on page 385.

data-type
Is one of the types in the following list. Use:

SMALLINT
For a small integer.

INTEGER or **INT**
For a large integer.

BIGINT
For a big integer.

FLOAT(*integer***)**
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 24 indicate single precision and the values 25 through 53 indicate double-precision.

You can also specify:

REAL	For single precision floating-point.
DOUBLE	For double-precision floating-point.
DOUBLE-PRECISION	For double-precision floating-point.
FLOAT	For double-precision floating-point.

DECIMAL*(precision-integer, scale-integer)* or **DEC***(precision-integer, scale-integer)*
For a decimal number. The first integer is the precision of the number; that is, the total number of digits; it may range from 1 to 31. The second integer is the scale of the number; that is, the number of digits to the right of the decimal point; it may range from 0 to the precision of the number.

If precision and scale are not specified, the default values of 5,0 are used. The words **NUMERIC** and **NUM** can be used as synonyms for **DECIMAL** and **DEC**.

CHARACTER*(integer)* or **CHAR***(integer)* or **CHARACTER** or **CHAR**
For a fixed-length character string of length *integer*, which may range from 1 to 254. If the length specification is omitted, a length of 1 character is assumed.

VARCHAR*(integer)*, or **CHARACTER VARYING***(integer)*, or **CHAR VARYING***(integer)*
For a varying-length character string of maximum length *integer*, which may range from 1 to 32 672.

LONG VARCHAR
For a varying-length character string with a maximum length of 32 700.

FOR BIT DATA
Specifies that the contents of the column are to be treated as bit (binary) data. During data exchange with other systems, code page conversions are not performed. Comparisons are done in binary, irrespective of the database collating sequence.

BLOB or BINARY LARGE OBJECT*(integer [K | M | G])*
For a binary large object string of the specified maximum length in bytes.

The length may be in the range of 1 byte to 2 147 483 647 bytes.

If *integer* by itself is specified, that is the maximum length.

If *integer K* (in either upper or lower case) is specified, the maximum length is 1 024 times *integer*. The maximum value for *integer* is 2 097 152. If a multiple of K, M or G that calculates out to 2 147 483 648 is specified, the actual value used is 2 147 483 647 (or 2 gigabytes minus 1 byte), which is the maximum length for a LOB column.

If *integer M* is specified, the maximum length is 1 048 576 times *integer*. The maximum value for *integer* is 2 048.

If *integer G* is specified, the maximum length is 1 073 741 824 times *integer*. The maximum value for *integer* is 2.

If the length specification is omitted, a length of 1 048 576 (1 megabyte) is assumed.

To create BLOB strings greater than 1 gigabyte, you must specify the NOT LOGGED option.

Any number of spaces is allowed between the integer and K, M, or G, and a space is not required. For example, all of the following are valid:

```
BLOB(50K)    BLOB(50 K)    BLOB (50   K)
```

CLOB or CHARACTER (CHAR) LARGE OBJECT*(integer [K | M | G])*

For a character large object string of the specified maximum length in bytes.

The meaning of the *integer K | M | G* is the same as for BLOB.

If the length specification is omitted, a length of 1 048 576 (1 megabyte) is assumed.

To create CLOB strings greater than 1 gigabyte, you must specify the NOT LOGGED option.

It is not possible to specify the FOR BIT DATA clause for CLOB columns. However, a CHAR FOR BIT DATA string can be assigned to a CLOB column, and a CHAR FOR BIT DATA string can be concatenated with a CLOB string.

DBCLOB*(integer [K | M | G])*

For a double-byte character large object string of the specified maximum length in double-byte characters.

The meaning of the *integer K | M | G* is similar to that for BLOB. The differences are that the number specified is the number of double-byte characters, and that the maximum size is 1 073 741 823 double-byte characters.

If the length specification is omitted, a length of 1 048 576 double-byte characters is assumed.

To create DBCLOB strings greater than 1 gigabyte, you must specify the NOT LOGGED option.

GRAPHIC*(integer)*

For a fixed-length graphic string of length *integer* which may range from 1 to 127. If the length specification is omitted, a length of 1 is assumed.

VARGRAPHIC*(integer)*

For a varying-length graphic string of maximum length *integer*, which may range from 1 to 16 336.

LONG VARGRAPHIC

For a varying-length graphic string with a maximum length of 16 350.

DATE
For a date.

TIME
For a time.

TIMESTAMP
For a timestamp.

DATALINK or **DATALINK***(integer)*
For a link to a data file stored outside the database.

The column in the table consists of "anchor values" that contain the reference information that is required to establish and maintain the link to the external data as well as an optional comment.

The length of a DATALINK column is 200 bytes. If *integer* is specified, it must be 200. If the length specification is omitted, a length of 200 bytes is assumed.

A DATALINK value is an encapsulated value with a set of built-in scalar functions. There is a function called DLVALUE to create a DATALINK value, and functions called DLNEWCOPY, DLPREVIOUSCOPY, and DLREPLACECONTENT that can also be used to construct a DATALINK value under special circumstances. (DLVALUE should be used to construct a regular DATALINK value.) The following functions can be used to extract attributes from a DATALINK value.

- DLCOMMENT
- DLLINKTYPE
- DLURLCOMPLETE
- DLURLCOMPLETEONLY
- DLURLCOMPLETEWRITE
- DLURLPATH
- DLURLPATHONLY
- DLURLPATHWRITE
- DLURLSCHEME
- DLURLSERVER

A DATALINK column has the following restrictions:

- The column cannot be part of any index. Therefore, it cannot be included as a column of a primary key or unique constraint (SQLSTATE 42962).
- The column cannot be a foreign key of a referential constraint (SQLSTATE 42830).

- A default value (WITH DEFAULT) cannot be specified for the column. If the column is nullable, the default for the column is NULL (SQLSTATE 42894).

distinct-type-name
For a user-defined type that is a distinct type. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL).

If a column is defined using a distinct type, then the data type of the column is the distinct type. The length and the scale of the column are respectively the length and the scale of the source type of the distinct type.

If a column defined using a distinct type is a foreign key of a referential constraint, then the data type of the corresponding column of the primary key must have the same distinct type.

structured-type-name
For a user-defined type that is a structured type. If a structured type name is specified without a schema name, the structured type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL, and by the CURRENT PATH register for dynamic SQL).

If a column is defined using a structured type, then the static data type of the column is the structured type. The column may include values with a dynamic type that is a subtype of *structured-type-name*.

A column defined using a structured type cannot be used in a primary key, unique constraint, foreign key, index key or partitioning key (SQLSTATE 42962).

If a column is defined using a structured type, and contains a reference-type attribute at any level of nesting, that reference-type attribute is unscoped. To use such an attribute in a dereference operation, it is necessary to specify a SCOPE explicitly, using a CAST specification.

If a column is defined using a structured type with an attribute of type DATALINK, or a distinct type sourced on DATALINK, this column can only be null. An attempt to use the constructor function for this type will return an error (SQLSTATE 428ED) and so no instance of this type can be inserted into the column.

REF (*type-name2***)**
For a reference to a typed table. If *type-name2* is specified without

a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). The underlying data type of the column is based on the representation data type specified in the REF USING clause of the CREATE TYPE statement for *type-name2* or the root type of the data type hierarchy that includes *type-name2*.

column-options
Defines additional options related to columns of the table.

NOT NULL
Prevents the column from containing null values.

If NOT NULL is not specified, the column can contain null values, and its default value is either the null value or the value provided by the WITH DEFAULT clause.

lob-options
Specifies options for LOB data types.

LOGGED
Specifies that changes made to the column are to be written to the log. The data in such columns is then recoverable with database utilities (such as RESTORE DATABASE). LOGGED is the default.

LOBs greater than 1 gigabyte cannot be logged (SQLSTATE 42993) and LOBs greater than 10 megabytes should probably not be logged.

NOT LOGGED
Specifies that changes made to the column are not to be logged.

NOT LOGGED has no effect on a commit or rollback operation; that is, the database's consistency is maintained even if a transaction is rolled back, regardless of whether or not the LOB value is logged. The implication of not logging is that during a roll forward operation, after a backup or load operation, the LOB data will be replaced by zeros for those LOB values that would have had log records replayed during the roll forward. During crash recovery, all committed changes and changes rolled back will reflect the expected results.

COMPACT
Specifies that the values in the LOB column should take up minimal disk space (free any extra disk pages in the last group used by the LOB value), rather than leave any leftover

space at the end of the LOB storage area that might facilitate subsequent append operations. Note that storing data in this way may cause a performance penalty in any append (length-increasing) operations on the column.

NOT COMPACT
Specifies some space for insertions to assist in future changes to the LOB values in the column. This is the default.

datalink-options
Specifies the options associated with a DATALINK data type.

LINKTYPE URL
This defines the type of link as a Uniform Resource Locator (URL).

NO LINK CONTROL
Specifies that there will not be any check made to determine that the file exists. Only the syntax of the URL will be checked. There is no database manager control over the file.

FILE LINK CONTROL
Specifies that a check should be made for the existence of the file. Additional options may be used to give the database manager further control over the file.

file-link-options
Additional options to define the level of database manager control of the file link.

INTEGRITY
Specifies the level of integrity of the link between a DATALINK value and the actual file.

ALL
Any file specified as a DATALINK value is under the control of the database manager and may NOT be deleted or renamed using standard file system programming interfaces.

READ PERMISSION
Specifies how permission to read the file specified in a DATALINK value is determined.

FS The read access permission is determined by the file system permissions. Such files can be accessed without retrieving the file name from the column.

DB
The read access permission is determined by the database. Access to the file will only be allowed by

passing a valid file access token, returned on retrieval of the DATALINK value from the table, in the open operation.

WRITE PERMISSION
Specifies how permission to write to the file specified in a DATALINK value is determined.

FS The write access permission is determined by the file system permissions. Such files can be accessed without retrieving the file name from the column.

BLOCKED
Write access is blocked. The file cannot be directly updated through any interface. An alternative mechanism must be used to cause updates to the information. For example, the file is copied, the copy updated, and then the DATALINK value updated to point to the new copy of the file.

ADMIN
The write permission is determined by the Data Links Manager. Write access to the file will only be allowed by passing a valid write token, returned on retrieval of the DATALINK value from the table, by using the DLURLCOMPLETEWRITE or DLURLPATHWRITE scalar function, in the open operation. This value can be specified only when READ PERMISSION DB is also specified.

The access privilege for a given linked file is defined and maintained in the Data Links Manager.

Once a file is opened for write with a valid write token by a user (the updater), other users can still open the file for read by using a valid read or write token. However, only the same updater can repeatedly open the file for write with the same write token. The same updater also needs the same write token to perform any subsequent read operation.

REQUIRING TOKEN FOR UPDATE
To complete the file update, the write token used to open and modify the file must be contained in the file reference specified during invocation of the scalar functions DLNEWCOPY or DLPREVIOUSCOPY in the SQL UPDATE statement.

NOT REQUIRING TOKEN FOR UPDATE
To complete the file update, a write token is not required in the file reference specified during invocation of the scalar functions DLNEWCOPY or DLPREVIOUSCOPY in the SQL UPDATE statement.

RECOVERY
Specifies whether or not DB2 will support point in time recovery of files referenced by values in this column.

YES
DB2 will support point in time recovery of files referenced by values in this column. This value can only be specified when INTEGRITY ALL and WRITE PERMISSION BLOCKED or WRITE PERMISSION ADMIN are also specified.

NO
Specifies that point in time recovery will not be supported.

ON UNLINK
Specifies the action taken on a file when a DATALINK value is changed or deleted (unlinked). Note that this is not applicable when WRITE PERMISSION FS is used.

RESTORE
Specifies that when a file is unlinked, the Data Links File Manager will attempt to return the file to the owner with the permissions that existed at the time the file was linked. In the case where the user is no longer registered with the file server, the file is assigned to a special predefined "dfmunknown" user ID. This can only be specified when INTEGRITY ALL and WRITE PERMISSION BLOCKED or WRITE PERMISSION ADMIN are also specified.

DELETE
Specifies that the file will be deleted when it is unlinked. This can only be specified when READ PERMISSION DB and WRITE PERMISSION BLOCKED or WRITE PERMISSION ADMIN are also specified.

MODE DB2OPTIONS
This mode defines a set of default file link options. The defaults defined by DB2OPTIONS are:

- INTEGRITY ALL

- READ PERMISSION FS
- WRITE PERMISSION FS
- RECOVERY NO

ON UNLINK is not applicable since WRITE PERMISSION FS is used.

SCOPE
Identifies the scope of the reference type column.

A scope must be specified for any column that is intended to be used as the left operand of a dereference operator or as the argument of the DEREF function. Specifying the scope for a reference type column may be deferred to a subsequent ALTER TABLE statement to allow the target table to be defined, usually in the case of mutually referencing tables.

typed-table-name
The name of a typed table. The table must already exist or be the same as the name of the table being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of values assigned to *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

typed-view-name
The name of a typed view. The view must already exist or be the same as the name of the view being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of values assigned to *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that was already specified within the same CREATE TABLE statement. (SQLSTATE 42710).

If this clause is omitted, an 18-character identifier that is unique among the identifiers of existing constraints defined on the table is generated by the system. (The identifier consists of ″SQL″ followed by a sequence of 15 numeric characters generated by a timestamp-based function.)

When used with a PRIMARY KEY or UNIQUE constraint, the *constraint-name* may be used as the name of an index that is created to support the constraint.

PRIMARY KEY
This 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.

A primary key cannot be specified if the table is a subtable (SQLSTATE 429B3) since the primary key is inherited from the supertable.

See PRIMARY KEY within the description of the *unique-constraint* below.

UNIQUE
This provides a shorthand method of defining a unique key 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.

A unique constraint cannot be specified if the table is a subtable (SQLSTATE 429B3) since unique constraints are inherited from the supertable.

See UNIQUE within the description of the *unique-constraint* below.

references-clause
This 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.

See *references-clause* under *referential-constraint* below.

CHECK (*check-condition***)**
This provides a shorthand method of defining a check constraint that applies to a single column. See CHECK (*check-condition*) below.

INLINE LENGTH *integer*
This option is only valid for a column defined using a structured type (SQLSTATE 42842) and indicates the maximum byte size of an instance of a structured type to store inline with the rest of the values in the row. Instances of structured types that cannot be stored inline are stored separately from the base table row, similar to the way that LOB values are handled. This takes place automatically.

The default INLINE LENGTH for a structured-type column is the inline length of its type (specified explicitly or by default in the

CREATE TYPE statement). If INLINE LENGTH of the structured type is less than 292, the value 292 is used for the INLINE LENGTH of the column.

Note: The inline lengths of subtypes are not counted in the default inline length, meaning that instances of subtypes may not fit inline unless an explicit INLINE LENGTH is specified at CREATE TABLE time to account for existing and future subtypes.

The explicit INLINE LENGTH value must be at least 292 and cannot exceed 32672 (SQLSTATE 54010).

COMPRESS SYSTEM DEFAULT
Specifies that system default values (that is, the default values used for the data types when no specific values are specified) are to be stored using minimal space. If the VALUE COMPRESSION clause is not specified, a warning is returned (SQLSTATE 01648) and system default values are not stored using minimal space.

Allowing system default values to be stored in this manner causes a slight performance penalty during insert and update operations on the column because of extra checking that is done.

The base data type must not be DATE, TIME, or TIMESTAMP (SQLSTATE 42842). If the base data type is a varying-length string, this clause is ignored. String values of length 0 are automatically compressed if a table has been set with VALUE COMPRESSION.

generated-column-spec

default-clause
Specifies a default value for the column.

WITH
An optional keyword.

DEFAULT
Provides a default value in the event a value is not supplied on INSERT or is specified as DEFAULT on INSERT or UPDATE. If a default value is not specified following the DEFAULT keyword, the default value depends on the data type of the column as shown in "ALTER TABLE".

If a column is defined as a DATALINK, then a default value cannot be specified (SQLSTATE 42613). The only possible default is NULL.

If the column is based on a column of a typed table, a specific default value must be specified when defining a default. A default value cannot be specified for the object identifier column of a typed table (SQLSTATE 42997).

If a column is defined using a distinct type, then the default value of the column is the default value of the source data type cast to the distinct type.

If a column is defined using a structured type, the *default-clause* cannot be specified (SQLSTATE 42842).

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column. If such a column is defined NOT NULL, then the column does not have a valid default.

default-values
Specific types of default values that can be specified are as follows.

constant
Specifies the constant as the default value 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 Chapter 3
- not be a floating-point constant unless the column is defined with a floating-point data type
- not have non-zero digits beyond the scale of the column data type if the constant is a decimal constant (for example, 1.234 cannot be the default for a DECIMAL(5,2) column)
- be expressed with no more than 254 characters including the quote characters, any introducer character such as the X for a hexadecimal constant, and characters from the fully qualified function name and parentheses when the constant is the argument of a *cast-function*.

datetime-special-register
Specifies the value of the datetime special register (CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be the data type that corresponds to the special register specified (for example, data type must be DATE when CURRENT DATE is specified).

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If CURRENT SCHEMA is specified, the data type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register.

USER
Specifies the value of the USER special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If USER is specified, the data type of the column must be a character string with a length not less than the length attribute of USER.

NULL
Specifies NULL as the default for the column. If NOT NULL was specified, DEFAULT NULL may be specified within the same column definition but will result in an error on any attempt to set the column to the default value.

cast-function
This form of a default value can only be used with columns defined as a distinct type, BLOB or datetime (DATE, TIME or TIMESTAMP) data type. For distinct type, with the exception of distinct types based on BLOB or datetime types, the name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type. For a distinct type based on a datetime type, where the default value is a constant, a function must be used and the name of the function must match the name of the source type of the distinct type with an implicit or explicit schema name of SYSIBM. For other datetime columns, the corresponding datetime function may also be used. For a BLOB or a distinct type based on BLOB, a function must be used and the name of the function must be BLOB with an implicit or explicit schema name of SYSIBM.

constant
Specifies a constant as the argument. The constant

must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. If the *cast-function* is BLOB, the constant must be a string constant.

datetime-special-register
Specifies CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The source type of the distinct type of the column must be the data type that corresponds to the specified special register.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register. The data type of the source type of the distinct type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register. If the cast-function is BLOB, the length attribute must be at least 8 bytes.

USER
Specifies the USER special register. The data type of the source type of the distinct type of the column must be a string data type with a length of at least 8 bytes. If the *cast-function* is BLOB, the length attribute must be at least 8 bytes.

If the value specified is not valid, an error (SQLSTATE 42894) is raised.

GENERATED
Indicates that DB2 generates values for the column. GENERATED must be specified if the column is to be considered an IDENTITY column.

ALWAYS
Indicates that DB2 will always generate a value for the column when a row is inserted into the table, or whenever the result value of the *generation-expression* may change. The result of the expression is stored in the table. GENERATED ALWAYS is the recommended value unless data propagation, or unload and reload operations are being done. GENERATED ALWAYS is the required value for generated columns.

BY DEFAULT
Indicates that DB2 will generate a value for the column when a row is inserted into the table, or updated,

specifying DEFAULT for the column, unless an explicit value is specified. BY DEFAULT is the recommended value when using data propagation or doing unload/reload.

Although not explicitly required, a unique, single-column index should be defined on the generated column to ensure uniqueness of the values.

AS IDENTITY
Specifies that the column is to be the identity column for this table. A table can only have a single IDENTITY column (SQLSTATE 428C1). The IDENTITY keyword can only be specified if the data type associated with the column is an exact numeric type 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 (SQLSTATE 42815). SMALLINT, INTEGER, BIGINT, or DECIMAL with a scale of zero, or a distinct type based on one of these types, are considered exact numeric types. By contrast, single- and double-precision floating points are considered approximate numeric data types. Reference types, even if represented by an exact numeric type, cannot be defined as identity columns.

An identity column is implicitly NOT NULL. An identity column cannot have a DEFAULT clause (SQLSTATE 42623).

START WITH *numeric-constant*
Specifies the first value for the identity column. This value can be any positive or negative value that could be assigned to this column (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA). The default is MINVALUE for ascending sequences, and MAXVALUE for descending sequences.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the identity column. This value can be any positive or negative value that could be assigned to this column (SQLSTATE 42815), and does not exceed the value of a large integer constant (SQLSTATE 42820), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA).

If this value is negative, this is a descending sequence. If this value is 0, or positive, this is an ascending sequence. 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. This value can be any positive or negative value that could be assigned to this column (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be less than or equal to the maximum value (SQLSTATE 42815).

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. This value can be any positive or negative value that could be assigned to this column (SQLSTATE 42815), without non-zero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be greater than or equal to the minimum value (SQLSTATE 42815).

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 sequence 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, DB2 may generate duplicate values for an identity column. Although not explicitly required, a unique, single-column index should be defined on the generated column to ensure uniqueness of the values, if unique values are desired. If a unique index exists on such an identity column and a non-unique value is generated, an error occurs (SQLSTATE 23505, SQLCODE -803).

NO CACHE or CACHE
Specifies whether to keep some pre-allocated values in memory for faster access. If a new value is needed for the identity column, and there are none available in the cache, then the end of the new cache block must be logged. However, when a new value is needed for the identity column, and there is an unused value in the cache, then the allocation of that identity value is faster, because no logging is necessary. This is a performance and tuning option.

NO CACHE
Specifies that values for the identity column are not to be pre-allocated. In an environment where identity sequence values are cached at more than one location, if identity values *must* be generated in order of request, the NO CACHE option must be used.

When this option is specified, the values of the identity column are not stored in the cache. In this case, every request for a new identity value results in synchronous I/O to the log.

CACHE *integer-constant*
Specifies how many values of the identity sequence are to be pre-allocated and kept in memory. When values are generated for the identity column, pre-allocating and storing values in the cache reduces synchronous I/O to the log.

If a new value is needed for the identity column and there are no unused values available in the cache, the allocation of the value involves waiting for I/O to the log. However, when a new value is needed for the identity column and there is an unused value in the cache, the allocation of that identity value can happen more quickly by avoiding the I/O to the log.

In the event of a database deactivation, either normally or due to a system failure, all cached sequence values that have not been used in committed statements are *lost*; that is, they will never be used. The value specified for the CACHE option is the maximum number of values for the identity column that could be lost in case of database deactivation. (If a database is not explicitly activated, using the ACTIVATE command or API, when the last application is disconnected from the database, an implicit deactivation occurs.)

The minimum value is 2 (SQLSTATE 42815). The default value is CACHE 20.

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 must be generated in order of request.

AS (*generation-expression***)**
Specifies that the definition of the column is based on an expression. (If the expression for a GENERATED ALWAYS column includes a user-defined external function, changing the executable for the function (such that the results change for given arguments) can result in inconsistent data. This can be avoided by using the SET INTEGRITY statement to force

the generation of new values.) The *generation-expression* cannot contain any of the following (SQLSTATE 42621):

- subqueries
- column functions
- dereference operations or DEREF functions
- user-defined or built-in functions that are non-deterministic
- user-defined functions using the EXTERNAL ACTION option
- user-defined functions defined with either CONTAINS SQL or READS SQL DATA
- host variables or parameter markers
- special registers
- references to columns defined later in the column list
- references to other generated columns

The data type for the column is based on the result data type of the *generation-expression*. A CAST specification can be used to force a particular data type and to provide a scope (for a reference type only). If *data-type* is specified, values are assigned to the column according to the appropriate assignment rules. A generated column is implicitly considered nullable, unless the NOT NULL column option is used. The data type of a generated column must be one for which equality is defined. This excludes columns of LONG VARCHAR, LONG VARGRAPHIC, LOB data types, DATALINKs, structured types, and distinct types based on any of these types (SQLSTATE 42962).

OID-column-definition
Defines the object identifier column for the typed table.

REF IS *OID-column-name* **USER GENERATED**
Specifies that an object identifier (OID) column is defined in the table as the first column. An OID is required for the root table of a table hierarchy (SQLSTATE 428DX). The table must be a typed table (the OF clause must be present) that is not a subtable (SQLSTATE 42613). The name for the column is defined as *OID-column-name* and cannot be the same as the name of any attribute of the structured type *type-name1* (SQLSTATE 42711). The column is defined with type REF(*type-name1*), NOT NULL and a system required unique index (with a default index name) is generated. This column is referred to as the *object identifier column* or *OID column*. The keywords USER GENERATED indicate that the initial value

for the OID column must be provided by the user when inserting a row. Once a row is inserted, the OID column cannot be updated (SQLSTATE 42808).

with-options
Defines additional options that apply to columns of a typed table.

column-name
Specifies the name of the column for which additional options are specified. The *column-name* must correspond to the name of a column of the table that is not also a column of a supertable (SQLSTATE 428DJ). A column name can only appear in one WITH OPTIONS clause in the statement (SQLSTATE 42613).

If an option is already specified as part of the type definition (in CREATE TYPE), the options specified here override the options in CREATE TYPE.

WITH OPTIONS *column-options*
Defines options for the specified column. See *column-options* described earlier. If the table is a subtable, primary key or unique constraints cannot be specified (SQLSTATE 429B3).

DATA CAPTURE
Indicates whether extra information for inter-database data replication is to be written to the log. This clause cannot be specified when creating a subtable (SQLSTATE 42613).

If the table is a typed table, then this option is not supported (SQLSTATE 428DH or 42HDR).

NONE
Indicates that no extra information will be logged.

CHANGES
Indicates that extra information regarding SQL changes to this table will be written to the log. This option is required if this table will be replicated and the Capture program is used to capture changes for this table from the log.

If the table is defined to allow data on a partition other than the catalog partition (multiple partition database partition group or database partition group with a partition other than the catalog partition), then this option is not supported (SQLSTATE 42997).

If the schema name (implicit or explicit) of the table is longer than 18 bytes, then this option is not supported (SQLSTATE 42997).

WITH RESTRICT ON DROP
Indicates that the table cannot be dropped, and that the table space that contains the table cannot be dropped.

FEDERATED
Specifies that the table being created references a federated object (a federated routine, a federated view, a nickname or an OLEDB table function). If an OLEDB table function or a nickname is directly or indirectly referenced in the fullselect, and the FEDERATED keyword is not specified, a warning (SQLSTATE 01639) is issued when the CREATE TABLE statement is invoked. However, the table will still be created.

Conversely, if an OLEDB table function or a nickname is not directly or indirectly referenced in the fullselect, and the FEDERATED keyword *is* specified, an error (SQLSTATE 429BA) is issued when the CREATE TABLE statement is invoked. The table will not be created.

NOT FEDERATED
Specifies that the table being created does not reference a nickname, a federated SQL routine, a federated view, or an OLEDB table function. If federated object is directly or indirectly referenced in the fullselect, and the NOT FEDERATED keyword is specified, an error (SQLSTATE 429BA) is issued when the CREATE TABLE statement is invoked. The table will not be created.

IN *tablespace-name1*
Identifies the table space in which the table will be created. The table space must exist, and be a REGULAR table space over which the authorization ID of the statement has USE privilege. If no other table space is specified, then all table parts will be stored in this table space. This clause cannot be specified when creating a subtable (SQLSTATE 42613), since the table space is inherited from the root table of the table hierarchy. If this clause is not specified, a table space for the table is determined as follows:

```
IF table space IBMDEFAULTGROUP over which the user
   has USE privilege exists with sufficient page size
     THEN choose it
 ELSE IF a table space over which the user has USE privilege
   exists with sufficient page size
   (see below when multiple table spaces qualify)
     THEN choose it
  ELSE issue an error (SQLSTATE 42727).
```

If more than one table space is identified by the ELSE IF condition, then choose the table space with the smallest sufficient page size over which the authorization ID of the statement has

USE privilege. When more than one table space qualifies, preference is given according to who was granted the USE privilege:

1. the authorization ID
2. a group to which the authorization ID belongs
3. PUBLIC

If more than one table space still qualifies, the final choice is made by the database manager.

Determination of the table space may change when:

- table spaces are dropped or created
- USE privileges are granted or revoked.

The sufficient page size of a table is determined by either the byte count of the row or the number of columns. See "Row Size" on page 385 for more information.

tablespace-options
Specifies the table space in which indexes and/or long column values will be stored. For details on types of table spaces, see "CREATE TABLESPACE".

INDEX IN *tablespace-name2*
Identifies the table space in which any indexes on the table will be created. This option is allowed only when the primary table space specified in the IN clause is a DMS table space. The specified table space must exist, must be a REGULAR or LARGE DMS table space over which the authorization ID of the statement has USE privilege, and must be in the same database partition group as *tablespace-name1* (SQLSTATE 42838).

Note that specifying which table space will contain a table's index can only be done when the table is created. The checking of USE privilege over the table space for the index is only carried out at table creation time. The database manager will not require that the authorization ID of a CREATE INDEX statement have USE privilege on the table space when an index is created later.

LONG IN *tablespace-name3*
Identifies the table space in which the values of any long columns (LONG VARCHAR, LONG VARGRAPHIC, LOB data types, distinct types with any of these as source types, or any columns defined with user-defined structured types with values that cannot be stored inline) will be stored. This option

is allowed only when the primary table space specified in the IN clause is a DMS table space. The table space must exist, must be a LARGE DMS table space over which the authorization ID of the statement has USE privilege, and must be in the same database partition group as *tablespace-name1* (SQLSTATE 42838).

Note that specifying which table space will contain a table's long and LOB columns can only be done when the table is created. The checking of USE privilege over the table space for the long and LOB columns is only carried out at table creation time. The database manager will not require that the authorization ID of an ALTER TABLE statement have USE privilege on the table space when a long or LOB column is added later.

PARTITIONING KEY (*column-name***,...)**

Specifies the partitioning key used when data in the table is partitioned. Each *column-name* must identify a column of the table and the same column must not be identified more than once. No column with data type that is a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, DATALINK, distinct type based on any of these types, or structured type may be used as part of a partitioning key (SQLSTATE 42962). A partitioning key cannot be specified for a table that is a subtable (SQLSTATE 42613), since the partitioning key is inherited from the root table in the table hierarchy.

If this clause is not specified, and this table resides in a multiple partition database partition group, then the partitioning key is defined as follows:

- if the table is a typed table, the object identifier column
- if a primary key is specified, the first column of the primary key is the partitioning key
- otherwise, the first column whose data type is not a LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK column, distinct type based on one of these types, or structured type column is the partitioning key.

If none of the columns satisfy the requirement of the default partitioning key, the table is created without one. Such tables are allowed only in table spaces defined on single-partition database partition groups.

For tables in table spaces defined on single-partition database partition groups, any collection of non-long type columns can be

used to define the partitioning key. If you do not specify this parameter, no partitioning key is created.

For restrictions related to the partitioning key, see 380.

USING HASHING
Specifies the use of the hashing function as the partitioning method for data distribution. This is the only partitioning method supported.

REPLICATED
Specifies that the data stored in the table is physically replicated on each database partition of the database partition group of the table space in which the table is defined. This means that a copy of all the data in the table exists on each of these database partitions. This option can only be specified for a materialized query table (SQLSTATE 42997).

VALUE COMPRESSION
Specifies that NULL and 0-length data values are to be stored more efficiently for most data types. This also determines the row format that is to be used. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

The 0-length data values for columns whose data type is BLOB, CLOB, DBCLOB, LONG VARCHAR, or LONG VARGRAPHIC are stored using minimal space. Each NULL value is stored without using an additional byte. The row format that is used to support this determines the byte counts for each data type, and tends to cause data fragmentation during updates. The new row format (specified for a column through the COMPRESS SYSTEM DEFAULT option) also allows system default values for the column to be stored more efficiently.

NOT LOGGED INITIALLY
Any changes made to the table by an Insert, Delete, Update, Create Index, Drop Index, or Alter Table operation in the same unit of work in which the table is created are not logged. For other considerations when using this option, see the "Notes" section of this statement.

All catalog changes and storage related information are logged, as are all operations that are done on the table in subsequent units of work.

Note: If non-logged activity occurs against a table that has the NOT LOGGED INITIALLY attribute activated, and if a statement fails (causing a rollback), or a ROLLBACK TO

SAVEPOINT is executed, the entire unit of work is rolled back (SQL1476N). Furthermore, the table for which the NOT LOGGED INITIALLY attribute was activated is marked inaccessible after the rollback has occurred, and can only be dropped. Therefore, the opportunity for errors within the unit of work in which the NOT LOGGED INITIALLY attribute is activated should be minimized.

OPTIONS

The OPTIONS clause can be used to override the remote name or the remote schema of the table being created. It must be used to specify the remote server.

REMOTE_SERVER *'value'*

Specifies the remote server on which the table is to be created.

REMOTE_SCHEMA *'value'*

Specifies a schema name for the remote table that is to be created. If a value is not specified, the local schema is used.

REMOTE_TABNAME *'value'*

Specifies an unqualified name for the remote table that is to be created. If a value is not specified, the local name is used.

unique-constraint

Defines a unique or primary key constraint. If the table has a partitioning key, then any unique or primary key must be a superset of the partitioning key. A unique or primary key constraint cannot be specified for a table that is a subtable (SQLSTATE 429B3). Primary or unique keys cannot be subsets of dimensions (SQLSTATE 429BE). If the table is a root table, the constraint applies to the table and all its subtables.

CONSTRAINT *constraint-name*

Names the primary key or unique constraint.

UNIQUE (*column-name***,...)**

Defines a unique key composed of the identified columns. The identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the table and the same column must not be identified more than once.

The number of identified columns must not exceed 16, and the sum of their stored lengths must not exceed 1024 (refer to "Byte Counts" on page 385 for the stored lengths). Unique keys with variable keyparts can have a size greater than 255 if the registry variable DB2_INDEX_2BYTEVARLEN is set to ON. No LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, distinct type based on one of these types, or structured type may be used as part of a unique key, even if the length attribute of the column is small enough to fit within the 1024-byte limit (SQLSTATE 54008).

The set of columns in the unique key cannot be the same as the set of columns in the primary key or another unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891.)

A unique constraint cannot be specified if the table is a subtable (SQLSTATE 429B3), because unique constraints are inherited from the supertable.

The description of the table as recorded in the catalog includes the unique key and its unique index. A unique index will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index will be the same as the *constraint-name* if this does not conflict with an existing index in the schema where the table is created. If the index name conflicts, the name will be SQL, followed by a character timestamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

PRIMARY KEY (*column-name***,...)**

Defines a primary key composed of the identified columns. The clause must not be specified more than once, and the identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the table, and the same column must not be identified more than once.

The number of identified columns must not exceed 16, and the sum of their stored lengths must not exceed 1024 (refer to "Byte Counts" on page 385 for the stored lengths). Primary keys with variable keyparts can have a size greater than 255 if the registry variable DB2_INDEX_2BYTEVARLEN is set to ON. No LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, distinct type based on one of these types, or structured type can be used as part of a primary key, even if the length attribute of the column is small enough to fit within the 1024-byte limit (SQLSTATE 54008).

The set of columns in the primary key cannot be the same as the set of columns in a unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891.)

Only one primary key can be defined on a table.

A primary key cannot be specified if the table is a subtable (SQLSTATE 429B3) since the primary key is inherited from the supertable.

The description of the table as recorded in the catalog includes the primary key and its primary index. A unique index will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index will be the same as the *constraint-name* if this does not conflict with an existing index in the

schema where the table is created. If the index name conflicts, the name will be SQL, followed by a character timestamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

If the table has a partitioning key, the columns of a *unique-constraint* must be a superset of the partitioning key columns; column order is unimportant.

referential-constraint
Defines a referential constraint.

CONSTRAINT *constraint-name*
Names the referential constraint.

FOREIGN KEY (*column-name***,...)**
Defines a referential constraint with the specified *constraint-name*.

Let T1 denote the object table of the statement. The foreign key of the referential constraint is composed of the identified columns. Each name in the list of column names must identify a column of T1 and the same column must not be identified more than once. The number of identified columns must not exceed 16 and the sum of their stored lengths must not exceed 1024 (refer to "Byte Counts" on page 385 for the stored lengths). Foreign keys can be defined on variable length columns whose length is greater than 255 bytes. No LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, distinct type based on one of these types, or structured type column may be used as part of a foreign key (SQLSTATE 42962). There must be the same number of foreign key columns as there are in the parent key and the data types of the corresponding columns must be compatible (SQLSTATE 42830). Two column descriptions are compatible if they have compatible data types (both columns are numeric, character strings, graphic, date/time, or have the same distinct type).

references-clause
Specifies the parent table and parent key for the referential constraint.

REFERENCES *table-name*
The table specified in a REFERENCES clause must identify a base table that is described in the catalog, but must not identify a catalog 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 ignored and a warning is issued (SQLSTATE 01543).

In the following discussion, let T2 denote the identified parent table, and let T1 denote the table being created (or altered). (T1 and T2 may be the same table).

The specified foreign key must have the same number of columns as the parent key of T2 and the description of the *n*th column of the foreign key must be comparable to the description of the *n*th column of that parent key. Datetime columns are not considered to be comparable to string columns for the purposes of this rule.

(*column-name***,...)**
The parent key of a 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 list of column names must match the set of columns (in any order) of the primary key or a unique constraint that exists on T2 (SQLSTATE 42890). If a column name list is not specified, then T2 must have a primary key (SQLSTATE 42888). Omission of the column name list is an implicit specification of the columns of that primary key in the sequence originally specified.

The referential constraint specified by a FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent.

rule-clause
Specifies what action to take on dependent tables.

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)
- RESTRICT
- CASCADE
- SET NULL

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.

- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

SET NULL must not be specified unless some column of the foreign key allows null values. Omission of the clause is an implicit specification of ON DELETE NO ACTION.

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 new 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. The NO ACTION and RESTRICT actions are treated identically. Thus, if T1 is a dependent of T3 in a relationship with a delete rule of *r*, the referential constraint cannot be defined when *r* is SET NULL if any of these conditions exist:

- T2 and T3 are the same table
- T2 is a descendant of T3 and the deletion of rows from T3 cascades to T2
- T3 is a descendant of T2 and the deletion of rows from T2 cascades to T3
- 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*.

In applying the above rules to referential constraints, in which either the parent table or the dependent table is a member of a typed table hierarchy, all the referential constraints that apply to any table in the respective hierarchies are taken into consideration.

ON UPDATE

Specifies what action is to take place on the dependent tables when a row of the parent table is updated. The clause is

optional. ON UPDATE NO ACTION is the default and ON UPDATE RESTRICT is the only alternative.

The difference between NO ACTION and RESTRICT is described in the "Notes" section of this statement.

constraint-attributes
Defines attributes associated with referential integrity or check constraints.

ENFORCED
The constraint is enforced by the database manager during normal operations, such as insert, update, or delete.

NOT ENFORCED
The constraint is not enforced by the database manager during normal operations, such as insert, update, or delete. This should only be specified if the table data is independently known to conform to the constraint.

ENABLE QUERY OPTIMIZATION
The constraint can be used for query optimization under appropriate circumstances.

DISABLE QUERY OPTIMIZATION
The constraint cannot be used for query optimization.

check-constraint
Defines a check constraint. A *check-constraint* is a *search-condition* that must evaluate to not false.

CONSTRAINT *constraint-name*
Names the check constraint.

CHECK (*check-condition***)**
Defines a check constraint. A *check-condition* is a *search-condition*, except as follows:

- A column reference must be to a column of the table being created.
- The *search-condition* cannot contain a TYPE predicate.
- It cannot contain any of the following (SQLSTATE 42621):
  - Subqueries
  - Dereference operations or DEREF functions where the scoped reference argument is other than the object identifier (OID) column
  - CAST specifications with a SCOPE clause
  - Column functions
  - Functions that are not deterministic
  - Functions defined to have an external action

- User-defined functions defined with either CONTAINS SQL or READS SQL DATA
- Host variables
- Parameter markers
- Special registers
- References to generated columns other than the identity column

If a check constraint is specified as part of a *column-definition*, a column reference can only be made to the same column. Check constraints specified as part of a table definition can have column references identifying columns previously defined in the CREATE TABLE statement. Check constraints are not checked for inconsistencies, duplicate conditions or equivalent conditions. Therefore, contradictory or redundant check constraints can be defined resulting in possible errors at execution time.

The check-condition "IS NOT NULL" can be specified; however, it is recommended that nullability be enforced directly, using the NOT NULL attribute of a column. For example, `CHECK (salary + bonus > 30000)` is accepted if salary is set to NULL, because CHECK constraints must be either satisfied or unknown, and in this case, salary is unknown. However, `CHECK (salary IS NOT NULL)` would be considered false and a violation of the constraint if salary is set to NULL.

Check constraints are enforced when rows in the table are inserted or updated. A check constraint defined on a table automatically applies to all subtables of that table.

Rules:

- The sum of the byte counts of the columns, including the inline lengths of all structured type columns, must not be greater than the row size limit that is based on the page size of the table space (SQLSTATE 54010). For more information, see "Byte Counts" on page 385. For typed tables, the byte count is applied to the columns of the root table of the table hierarchy, and every additional column introduced by every subtable in the table hierarchy (additional subtable columns must be considered nullable for byte count purposes, even if defined as not nullable). There is also an additional 4 bytes of overhead to identify the subtable to which each row belongs.
- The number of columns in a table cannot exceed 1 012 (SQLSTATE 54011). For typed tables, the total number of attributes of the types of all of the subtables in the table hierarchy cannot exceed 1010.
- An object identifier column of a typed table cannot be updated (SQLSTATE 42808).

- Any unique or primary key constraint defined on the table must be a superset of the partitioning key (SQLSTATE 42997).
- The following table provides the supported combinations of DATALINK options in the *file-link-options* (SQLSTATE 42613). WRITE PERMISSION ADMIN can only combine with READ PERMISSION DB. (Other combinations in the RECOVERY and the ON UNLINK clause are supported.)

Table 5. Valid DATALINK File Control Option Combinations. Any combination that cannot be found in this table is not supported, and results in SQLSTATE 42613.

INTEGRITY	READ PERMISSION	WRITE PERMISSION	RECOVERY	ON UNLINK
ALL	FS	FS	NO	Not applicable
ALL	FS	BLOCKED	NO	RESTORE
ALL	FS	BLOCKED	YES	RESTORE
ALL	DB	BLOCKED	NO	RESTORE
ALL	DB	BLOCKED	NO	DELETE
ALL	DB	BLOCKED	YES	RESTORE
ALL	DB	BLOCKED	YES	DELETE
ALL	DB	ADMIN	NO	RESTORE
ALL	DB	ADMIN	NO	DELETE
ALL	DB	ADMIN	YES	RESTORE
ALL	DB	ADMIN	YES	DELETE

The following rules only apply to partitioned databases.

- Tables composed only of columns with types LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, distinct type based on one of these types, or structured type can only be created in table spaces defined on single-partition database partition groups.
- The partitioning key definition of a table in a table space defined on a multiple partition database partition group cannot be altered.
- The partitioning key column of a typed table must be the OID column.

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - The CONSTRAINT keyword can be omitted from a *column-definition* defining a references-clause.
    - *constraint-name* can be specified following FOREIGN KEY (without the CONSTRAINT keyword).

    - SUMMARY can optionally be specified after CREATE.
  - For compatibility with previous versions of DB2, and for consistency:
    - A comma can be used to separate multiple options in the *identity-attributes* clause.
  - For compatibility with DB2 UDB for OS/390 and z/OS:
    - The following syntax is accepted as the default behavior:
      - CCSID ASCII
      - CCSID UNICODE
      - IN database-name.tablespace-name
      - IN DATABASE database-name
      - FOR MIXED DATA
      - FOR SBCS DATA
    - The following syntax is also supported:
      - NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER
- Creating a table with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- If a foreign key is specified:
  - All packages with a delete usage on the parent table are invalidated.
  - All packages with an update usage on at least one column in the parent key are invalidated.
- Creating a subtable causes invalidation of all packages that depend on any table in table hierarchy.
- VARCHAR and VARGRAPHIC columns that are greater than 4 000 and 2 000 respectively should not be used as input parameters in functions in SYSFUN schema. Errors will occur when the function is invoked with an argument value that exceeds these lengths (SQLSTATE 22001).
- Identity columns are not supported in a database with multiple partitions (SQLSTATE 42997). An identity column cannot be created if more than one partition for the database exists. A database that includes any identity columns cannot be started with more than one partition.
- The use of NO ACTION or RESTRICT as delete or update rules for referential constraints determines when the constraint is enforced. A delete or update rule of RESTRICT is enforced *before* all other constraints, including those referential constraints with modifying rules such as CASCADE or SET NULL. A delete or update rule of NO ACTION is enforced *after* other referential constraints. There are very few cases where this can make a difference during a delete or update. One example where

different behavior is evident involves the deletion of rows from a view that is defined as a UNION ALL of related tables.

```
Table T1 is a parent of table T3; delete rule as noted below.
Table T2 is a parent of table T3; delete rule CASCADE.

CREATE VIEW V1 AS SELECT * FROM T1 UNION ALL SELECT * FROM T2

DELETE FROM V1
```

If table T1 is a parent of table T3 with a delete rule of RESTRICT, a restrict violation will be raised (SQLSTATE 23001) if there are any child rows for parent keys of T1 in T3.

If table T1 is a parent of table T3 with a delete rule of NO ACTION, the child rows may be deleted by the delete rule of CASCADE when deleting rows from T2 before the NO ACTION delete rule is enforced for the deletes from T1. If deletes from T2 did not result in deleting all child rows for parent keys of T1 in T3, then a constraint violation will be raised (SQLSTATE 23504).

Note that the SQLSTATE returned is different depending on whether the delete or update rule is RESTRICT or NO ACTION.

- For tables in table spaces defined on multiple partition database partition groups, table collocation should be considered in choosing the partitioning keys. Following is a list of items to consider:
  - The tables must be in the same database partition group for collocation. The table spaces may be different, but must be defined in the same database partition group.
  - The partitioning keys of the tables must have the same number of columns, and the corresponding key columns must be partition compatible for collocation.
  - The choice of partitioning key also has an impact on performance of joins. If a table is frequently joined with another table, you should consider the joining column(s) as a partitioning key for both tables.
- The NOT LOGGED INITIALLY clause can not be used when DATALINK columns with the FILE LINK CONTROL attribute are present in the table (SQLSTATE 42613).
- The NOT LOGGED INITIALLY option is useful for situations where a large result set needs to be created with data from an alternate source (another table or a file) and recovery of the table is not necessary. Using this option will save the overhead of logging the data. The following considerations apply when this option is specified:
  - When the unit of work is committed, all changes that were made to the table during the unit of work are flushed to disk.

  - When you run the rollforward utility and it encounters a log record that indicates that a table in the database was either populated by the Load utility or created with the NOT LOGGED INITIALLY option, the table will be marked as unavailable. The table will be dropped by the rollforward utility if it later encounters a DROP TABLE log. Otherwise, after the database is recovered, an error will be issued if any attempt is made to access the table (SQLSTATE 55019). The only operation permitted is to drop the table.
  - Once such a table is backed up as part of a database or table space back up, recovery of the table becomes possible.

- A REFRESH DEFERRED system-maintained materialized query table defined with ENABLE QUERY OPTIMIZATION can be used to optimize the processing of queries if CURRENT REFRESH AGE is set to ANY and CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is set such that it includes system-maintained materialized query tables. A REFRESH DEFERRED user-maintained materialized query table defined with ENABLE QUERY OPTIMIZATION can be used to optimize the processing of queries if CURRENT REFRESH AGE is set to ANY and CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is set such that it includes user-maintained materialized query tables. A REFRESH IMMEDIATE materialized query table defined with ENABLE QUERY OPTIMIZATION is always considered for optimization. For this optimization to be able to use a REFRESH DEFERRED or a REFRESH IMMEDIATE materialized query table, the fullselect must conform to certain rules in addition to those already described. The fullselect must:
  - be a subselect with a GROUP BY clause or a subselect with a single table reference
  - not include DISTINCT anywhere in the select list
  - not include any special registers
  - not include functions that are not deterministic.

  If the query specified when creating a materialized query table does not conform to these rules, a warning is returned (SQLSTATE 01633).
- If a materialized query table is defined with REFRESH IMMEDIATE, or a staging table is defined with PROPAGATE IMMEDIATE, it is possible for an error to occur when attempting to apply the change resulting from an insert, update or delete operation on an underlying table. The error will cause the failure of the insert, update or delete of the underlying table.
- A referential constraint may be defined in such a way that either the parent table or the dependent table is a part of a table hierarchy. In such a case, the effect of the referential constraint is as follows:
  1. Effects of INSERT, UPDATE, and DELETE statements:
     - If a referential constraint exists, in which PT is a parent table and DT is a dependent table, the constraint ensures that for each row of DT

(or any of its subtables) that has a non-null foreign key, a row exists in PT (or one of its subtables) with a matching parent key. This rule is enforced against any action that affects a row of PT or DT, regardless of how that action is initiated.

2. Effects of DROP TABLE statements:
   - for referential constraints in which the dropped table is the parent table or dependent table, the constraint is dropped
   - for referential constraints in which a supertable of the dropped table is the parent table the rows of the dropped table are considered to be deleted from the supertable. The referential constraint is checked and its delete rule is invoked for each of the deleted rows.
   - for referential constraints in which a supertable of the dropped table is the dependent table, the constraint is not checked. Deletion of a row from a dependent table cannot result in violation of a referential constraint.

- ***Privileges:*** When any table is created, the definer of the table is granted CONTROL privilege. When a subtable is created, the SELECT privilege that each user or group has on the immediate supertable is automatically granted on the subtable with the table definer as the grantor.
- ***Row Size:*** The maximum number of bytes allowed in the row of a table is dependent on the page size of the table space in which the table is created (*tablspace-name1*). The following list shows the row size limit and number of columns limit associated with each table space page size.

Table 6. Limits for Number of Columns and Row Size in Each Table Space Page Size

Page Size	Row Size Limit	Column Count Limit
4K	4 005	500
8K	8 101	1 012
16K	16 293	1 012
32K	32 677	1 012

The actual number of columns for a table may be further limited by the following formula:

```
Total Columns * 8 + Number of LOB Columns * 12 +
  Number of Datalink Columns * 28 <= row size limit for page size.
```

- ***Byte Counts:*** The following table contains the byte counts of columns by data type for columns that do not allow null values. In tables without value compression, each column that allows nulls requires an additional byte.

  If a table is based on a structured type, an additional 4 bytes of overhead is reserved to identify rows of subtables, regardless of whether or not subtables are defined. Additional subtable columns must be considered nullable for byte count purposes, even if defined as not nullable.

When calculating stored lengths, to ensure that the 1024-byte limit is not exceeded in an index or in constraints (note that constraints are enforced through indexes), the overhead is 2 bytes instead of 4 bytes.

Table 7. Byte Counts of Columns by Data Type

Data type	Byte count when VALUE COMPRESSION is activated for the table	Byte count when VALUE COMPRESSION is not activated, either implicitly or explicitly, for the table; if the column is nullable, add 1 to the indicated byte count
ROW OVERHEAD	2	0
INTEGER	6	4
SMALLINT	4	2
BIGINT	10	8
REAL	6	4
DOUBLE	10	8
DECIMAL	The integral part of (p/2)+3, where p is the precision	The integral part of (p/2)+1, where p is the precision
CHAR(n)	n+2	n
VARCHAR(n)	n+2	n+4 (within a table); n+2 (within an index)
LONG VARCHAR	22	24
GRAPHIC(n)	n*2+2	n*2
VARGRAPHIC(n)	(n*2)+2	(n*2)+4 (within a table); (n*2)+2 (within an index)
LONG VARGRAPHIC	22	24
DATE	6	4
TIME	5	3
TIMESTAMP	12	10
DATALINK(n)	n+52	n+54
Maximum LOB length 1024	70[1]	72
Maximum LOB length 8192	94	96
Maximum LOB length 65 536	118	120
Maximum LOB length 524 000	142	144

Table 7. Byte Counts of Columns by Data Type (continued)

Data type	Byte count when VALUE COMPRESSION is activated for the table	Byte count when VALUE COMPRESSION is not activated, either implicitly or explicitly, for the table; if the column is nullable, add 1 to the indicated byte count
Maximum LOB length 4 190 000	166	168
Maximum LOB length 134 000 000	198	200
Maximum LOB length 536 000 000	222	224
Maximum LOB length 1 070 000 000	254	256
Maximum LOB length 1 470 000 000	278	280
Maximum LOB length 2 147 483 647	314	316

[1] Each LOB value has a *LOB descriptor* in the base record that points to the location of the actual value. The size of the descriptor varies according to the maximum length defined for the column.

For a *distinct type*, the byte count is equivalent to the length of the source type of the distinct type. For a *reference type*, the byte count is equivalent to the length of the built-in data type on which the reference type is based. For a *structured type*, the byte count is equivalent to the INLINE LENGTH + 4. The INLINE LENGTH is the value specified (or implicitly calculated) for the column in the *column-options* clause.

- ***Dimension Columns:*** Because each distinct value of a dimension column is assigned to a different block of the table, clustering on an expression may be desirable, such as "INTEGER(ORDER_DATE)/100". In this case, a generated column can be defined for the table, and this generated column may then be used in the ORGANIZE BY DIMENSIONS clause. If the expression is monotonic with respect to a column of the table, DB2 may use the dimension index to satisfy range predicates on that column. For example, if the expression is simply *column-name + some-positive-constant*, it is monotonic increasing. User-defined functions, certain built-in functions, and using more than one column in an expression, prevent monotonicity or its detection.

  Dimensions involving generated columns whose expressions are non-monotonic, or whose monotonicity cannot be determined, can still be

created, but range queries along slice or cell boundaries of these dimensions are not supported. Equality and IN predicates *can* be processed by slices or cells.

A generated column is monotonic if the following is true with respect to the generating function, fn:

– Monotonic increasing.

  For every possible pair of values x1 and x2, if x2>x1, then fn(x2)>fn(x1). For example:

  ```
  SALARY - 10000
  ```

– Monotonic decreasing.

  For every possible pair of values x1 and x2, if x2>x1, then fn(x2)<fn(x1). For example:

  ```
  -SALARY
  ```

– Monotonic non-decreasing.

  For every possible pair of values x1 and x2, if x2>x1, then fn(x2)>=fn(x1). For example:

  ```
  SALARY/1000
  ```

– Monotonic non-increasing.

  For every possible pair of values x1 and x2, if x2>x1, then fn(x2)<=fn(x1). For example:

  ```
  -SALARY/1000
  ```

The expression ″PRICE*DISCOUNT″ is not monotonic, because it involves more than one column of the table.

Examples:

Example 1: Create table TDEPT in the DEPARTX table space. DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT are column names. CHAR means the column will contain character data. NOT NULL means that the column cannot contain a null value. VARCHAR means the column will contain varying-length character data. The primary key consists of the column DEPTNO.

```
CREATE TABLE TDEPT
  (DEPTNO   CHAR(3)     NOT NULL,
   DEPTNAME VARCHAR(36) NOT NULL,
   MGRNO    CHAR(6),
   ADMRDEPT CHAR(3)     NOT NULL,
   PRIMARY KEY(DEPTNO))
IN DEPARTX
```

Example 2: Create table PROJ in the SCHED table space. PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF, PRSTDATE, PRENDATE, and MAJPROJ are column names. CHAR means the column will contain character

data. DECIMAL means the column will contain packed decimal data. 5,2 means the following: 5 indicates the number of decimal digits, and 2 indicates the number of digits to the right of the decimal point. NOT NULL means that the column cannot contain a null value. VARCHAR means the column will contain varying-length character data. DATE means the column will contain date information in a three-part format (year, month, and day).

```
CREATE TABLE PROJ
  (PROJNO   CHAR(6)      NOT NULL,
   PROJNAME VARCHAR(24)  NOT NULL,
   DEPTNO   CHAR(3)      NOT NULL,
   RESPEMP  CHAR(6)      NOT NULL,
   PRSTAFF  DECIMAL(5,2)         ,
   PRSTDATE DATE                 ,
   PRENDATE DATE                 ,
   MAJPROJ  CHAR(6)      NOT NULL)
IN SCHED
```

Example 3: Create a table called EMPLOYEE_SALARY where any unknown salary is considered 0. No table space is specified, so that the table will be created in a table space selected by the system based on the rules described for the *IN tablespace-name1* clause.

```
CREATE TABLE EMPLOYEE_SALARY
  (DEPTNO   CHAR(3)      NOT NULL,
   DEPTNAME VARCHAR(36)  NOT NULL,
   EMPNO    CHAR(6)      NOT NULL,
   SALARY   DECIMAL(9,2) NOT NULL WITH DEFAULT)
```

Example 4: Create distinct types for total salary and miles and use them for columns of a table created in the default table space. In a dynamic SQL statement assume the CURRENT SCHEMA special register is JOHNDOE and the CURRENT PATH is the default (″SYSIBM″,″SYSFUN″,″JOHNDOE″).

If a value for SALARY is not specified it must be set to 0 and if a value for LIVING_DIST is not specified it must to set to 1 mile.

```
CREATE DISTINCT TYPE JOHNDOE.T_SALARY AS INTEGER WITH COMPARISONS

CREATE DISTINCT TYPE JOHNDOE.MILES  AS FLOAT WITH COMPARISONS

CREATE TABLE EMPLOYEE
  (ID          INTEGER NOT NULL,
   NAME        CHAR (30),
   SALARY      T_SALARY NOT NULL WITH DEFAULT,
   LIVING_DIST MILES    DEFAULT MILES(1) )
```

Example 5: Create distinct types for image and audio and use them for columns of a table. No table space is specified, so that the table will be created in a table space selected by the system based on the rules described for the IN *tablespace-name1* clause. Assume the CURRENT PATH is the default.

```
CREATE DISTINCT TYPE IMAGE AS BLOB (10M)

CREATE DISTINCT TYPE AUDIO AS BLOB (1G)

CREATE TABLE PERSON
  (SSN    INTEGER NOT NULL,
   NAME   CHAR (30),
   VOICE  AUDIO,
   PHOTO  IMAGE)
```

Example 6: Create table EMPLOYEE in the HUMRES table space. The constraints defined on the table are the following:

- The values of department number must lie in the range 10 to 100.
- The job of an employee can only be either 'Sales', 'Mgr' or 'Clerk'.
- Every employee that has been with the company since 1986 must make more than $40,500.

Note: If the columns included in the check constraints are nullable they could also be NULL.

```
CREATE TABLE EMPLOYEE
  (ID           SMALLINT NOT NULL,
   NAME         VARCHAR(9),
   DEPT         SMALLINT CHECK (DEPT BETWEEN 10 AND 100),
   JOB          CHAR(5) CHECK (JOB IN ('Sales','Mgr','Clerk')),
   HIREDATE     DATE,
   SALARY       DECIMAL(7,2),
   COMM         DECIMAL(7,2),
   PRIMARY KEY (ID),
   CONSTRAINT  YEARSAL CHECK (YEAR(HIREDATE) > 1986
     OR SALARY > 40500)
  )
  IN HUMRES
```

Example 7: Create a table that is wholly contained in the PAYROLL table space.

```
CREATE TABLE EMPLOYEE .....
  IN PAYROLL
```

Example 8: Create a table with its data part in ACCOUNTING and its index part in ACCOUNT_IDX.

```
CREATE TABLE SALARY.....
  IN ACCOUNTING INDEX IN ACCOUNT_IDX
```

Example 9: Create a table and log SQL changes in the default format.

```
CREATE TABLE SALARY1 .....
```

or

```
CREATE TABLE SALARY1 .....
  DATA CAPTURE NONE
```

Example 10: Create a table and log SQL changes in an expanded format.

```
CREATE TABLE SALARY2 .....
  DATA CAPTURE CHANGES
```

Example 11: Create a table EMP_ACT in the SCHED table space. EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, and EMENDATE are column names. Constraints defined on the table are:

- The value for the set of columns, EMPNO, PROJNO, and ACTNO, in any row must be unique.
- The value of PROJNO must match an existing value for the PROJNO column in the PROJECT table and if the project is deleted all rows referring to the project in EMP_ACT should also be deleted.

```
CREATE TABLE EMP_ACT
  (EMPNO       CHAR(6) NOT NULL,
   PROJNO      CHAR(6) NOT NULL,
   ACTNO       SMALLINT NOT NULL,
   EMPTIME     DECIMAL(5,2),
   EMSTDATE    DATE,
   EMENDATE    DATE,
   CONSTRAINT EMP_ACT_UNIQ UNIQUE (EMPNO,PROJNO,ACTNO),
   CONSTRAINT FK_ACT_PROJ FOREIGN KEY (PROJNO)
                          REFERENCES PROJECT (PROJNO) ON DELETE CASCADE
  )
  IN SCHED
```

A unique index called EMP_ACT_UNIQ is automatically created in the same schema to enforce the unique constraint.

Example 12: Create a table that is to hold information about famous goals for the ice hockey hall of fame. The table will list information about the player who scored the goal, the goaltender against who it was scored, the date and place, and a description. When available, it will also point to places where newspaper articles about the game are stored and where still and moving pictures of the goal are stored. The newspaper articles are to be linked so they cannot be deleted or renamed but all existing display and update applications must continue to operate. The still pictures and movies are to be linked with access under complete control of DB2. The still pictures are to have recovery and are to be returned to their original owner if unlinked. The movie pictures are not to have recovery and are to be deleted if unlinked. The description column and the three DATALINK columns are nullable.

```
CREATE TABLE HOCKEY_GOALS
  ( BY_PLAYER      VARCHAR(30)   NOT NULL,
    BY_TEAM        VARCHAR(30)   NOT NULL,
    AGAINST_PLAYER VARCHAR(30)   NOT NULL,
    AGAINST_TEAM   VARCHAR(30)   NOT NULL,
```

```
  DATE_OF_GOAL   DATE          NOT NULL,
  DESCRIPTION    CLOB(5000),
  ARTICLES       DATALINK  LINKTYPE URL FILE LINK CONTROL MODE DB2OPTIONS,
  SNAPSHOT       DATALINK  LINKTYPE URL FILE LINK CONTROL
                           INTEGRITY ALL
                           READ PERMISSION DB WRITE PERMISSION BLOCKED
                           RECOVERY YES ON UNLINK RESTORE,
  MOVIE          DATALINK  LINKTYPE URL FILE LINK CONTROL
                           INTEGRITY ALL
                           READ PERMISSION DB WRITE PERMISSION BLOCKED
                           RECOVERY NO ON UNLINK DELETE )
```

Example 13: Suppose an exception table is needed for the EMPLOYEE table. One can be created using the following statement.

```
CREATE TABLE EXCEPTION_EMPLOYEE AS
  (SELECT EMPLOYEE.*,
    CURRENT TIMESTAMP AS TIMESTAMP,
    CAST ('' AS CLOB(32K)) AS MSG
  FROM EMPLOYEE
  ) DEFINITION ONLY
```

Example 14: Given the following table spaces with the indicated attributes:

```
TBSPACE            PAGESIZE    USER   USERAUTH
------------------ ----------- ------ --------
DEPT4K                    4096 BOBBY  Y
PUBLIC4K                  4096 PUBLIC Y
DEPT8K                    8192 BOBBY  Y
DEPT8K                    8192 RICK   Y
PUBLIC8K                  8192 PUBLIC Y
```

- If RICK creates the following table, it is placed in table space PUBLIC4K since the byte count is less than 4005; but if BOBBY creates the same table, it is placed in table space DEPT4K, since BOBBY has USE privilege because of an explicit grant:

  ```
  CREATE TABLE DOCUMENTS
    (SUMMARY    VARCHAR(1000),
     REPORT     VARCHAR(2000))
  ```

- If BOBBY creates the following table, it is placed in table space DEPT8K since the byte count is greater than 4005, and BOBBY has USE privilege because of an explicit grant. However, if DUNCAN creates the same table, it is placed in table space PUBLIC8K, since DUNCAN has no specific privileges:

  ```
  CREATE TABLE CURRICULUM
    (SUMMARY    VARCHAR(1000),
     REPORT     VARCHAR(2000),
     EXERCISES  VARCHAR(1500))
  ```

Example 15: Create a table with a LEAD column defined with the structured type EMP. Specify an INLINE LENGTH of 300 bytes for the LEAD column,

indicating that any instances of LEAD that cannot fit within the 300 bytes are stored outside the table (separately from the base table row, similar to the way LOB values are handled).

```
CREATE TABLE PROJECTS (PID INTEGER,
  LEAD EMP INLINE LENGTH 300,
  STARTDATE DATE,
            ...)
```

Example 16: Create a table DEPT with five columns named DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, and LOCATION. Column DEPT is to be defined as an IDENTITY column such that DB2 will always generate a value for it. The values for the DEPT column should begin with 500 and increment by 1.

```
CREATE TABLE DEPT
  (DEPTNO SMALLINT NOT NULL
    GENERATED ALWAYS AS IDENTITY
    (START WITH 500, INCREMENT BY 1),
  DEPTNAME VARCHAR (36) NOT NULL,
  MGRNO CHAR(6),
  ADMRDEPT SMALLINT NOT NULL,
  LOCATION CHAR(30))
```

Related reference:

- "Subselect" in the *SQL Reference, Volume 1*
- "ALTER TABLE" on page 41
- "CREATE TABLESPACE" on page 395
- "DECLARE GLOBAL TEMPORARY TABLE" on page 488
- "Assignments and comparisons" in the *SQL Reference, Volume 1*
- "Partition-compatible data types" in the *SQL Reference, Volume 1*

Related samples:

- "dtudt.c -- How to create, use, and drop user-defined distinct types. (CLI)"
- "tbconstr.c -- How to work with constraints associated with tables (CLI)"
- "tbcreate.c -- How to create, alter and drop tables (CLI)"
- "dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)"
- "tbconstr.sqc -- How to create, use, and drop constraints (C)"
- "tbcreate.sqc -- How to create and drop tables (C)"
- "tbident.sqc -- How to use identity columns (C)"
- "tbtrig.sqc -- How to use a trigger on a table (C)"
- "dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)"
- "tbconstr.sqC -- How to create, use, and drop constraints (C++)"
- "tbcreate.sqC -- How to create and drop tables (C++)"

- "tbtrig.sqC -- How to use a trigger on a table (C++)"
- "DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)"
- "TbConstr.java -- How to create, use and drop constraints (JDBC)"
- "TbCreate.java -- How to create and drop tables (JDBC)"
- "TbGenCol.java -- How to use generated columns (JDBC)"
- "TbIdent.java -- How to use Identity Columns (JDBC)"
- "TbTrig.java -- How to use triggers (JDBC)"
- "DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)"
- "TbConstr.sqlj -- How to create, use and drop constraints (SQLj)"
- "TbCreate.sqlj -- How to create and drop tables (SQLj)"
- "TbIdent.sqlj -- How to use Identity Columns (SQLj)"
- "TbTrig.sqlj -- How to use triggers (SQLj)"
- "impexp.sqb -- Export and import tables with table data (MF COBOL)"

CREATE TABLESPACE

The CREATE TABLESPACE statement creates a new tablespace within the database, assigns containers to the tablespace, and records the tablespace definition and attributes in the catalog.

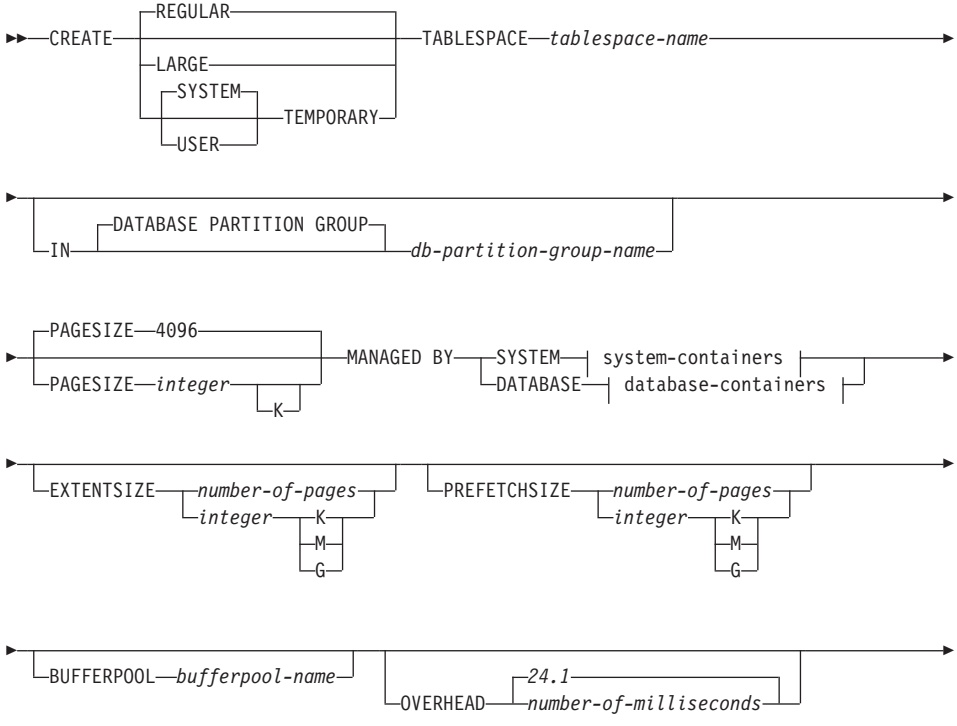
Invocation:

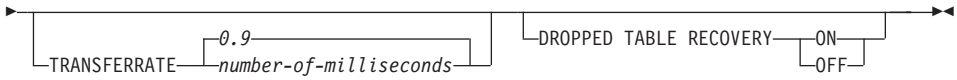
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

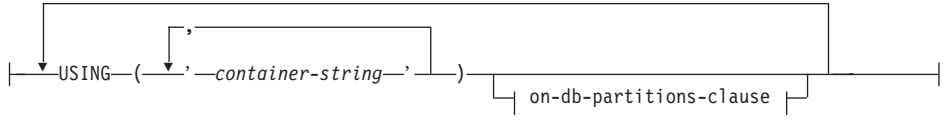
The authorization ID of the statement must have SYSCTRL or SYSADM authority.

Syntax:

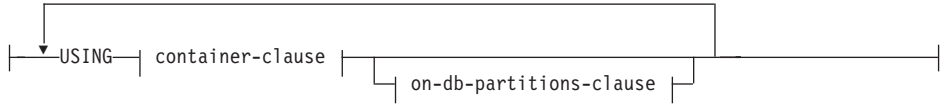




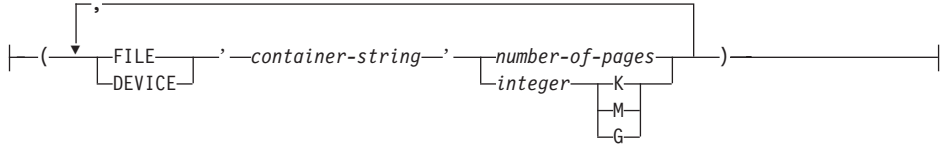
system-containers:



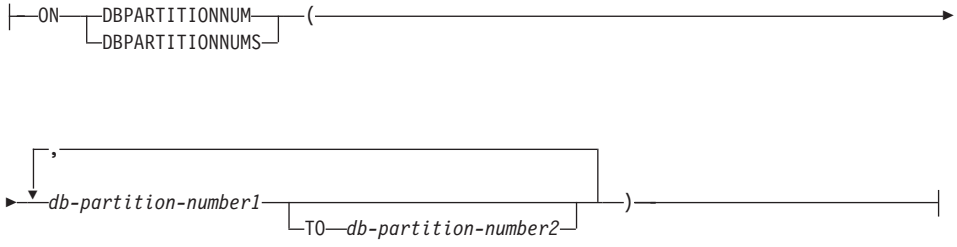
database-containers:



container-clause:



on-db-partitions-clause:



Description:

REGULAR
Stores all data except for temporary tables.

LARGE
Stores long or LOB table columns. It can also store structured type columns or index data. The tablespace must be a DMS tablespace.

SYSTEM TEMPORARY
Stores temporary tables (work areas used by the database manager to perform operations such as sorts or joins). The keyword SYSTEM is optional. Note that a database must always have at least one SYSTEM TEMPORARY tablespace, as temporary tables can only be stored in such a tablespace. A temporary tablespace is created automatically when a database is created.

USER TEMPORARY
Stores declared global temporary tables. Note that no user temporary tablespaces exist when a database is created. At least one user temporary tablespace should be created with appropriate USE privileges, to allow definition of declared temporary tables.

tablespace-name
Names the tablespace. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *tablespace-name* must not identify a tablespace that already exists in the catalog (SQLSTATE 42710). The *tablespace-name* must not begin with the characters SYS (SQLSTATE 42939).

IN DATABASE PARTITION GROUP *db-partition-group-name*
Specifies the database partition group for the tablespace. The database partition group must exist. The only database partition group that can be specified when creating a SYSTEM TEMPORARY tablespace is IBMTEMPGROUP. The DATABASE PARTITION GROUP keywords are optional.

If the database partition group is not specified, the default database partition group (IBMDEFAULTGROUP) is used for REGULAR, LARGE, and USER TEMPORARY tablespaces. For SYSTEM TEMPORARY tablespaces, the default database partition group IBMTEMPGROUP is used.

PAGESIZE *integer* **[K]**
Defines the size of pages used for the tablespace. The valid values for *integer* without the suffix K are 4 096 or 8 192, 16 384, or 32 768. The valid values for *integer* with the suffix K are 4 or 8, 16, or 32. An error occurs if the page size is not one of these values (SQLSTATE 428DE) or the page size is not the same as the page size of the bufferpool associated with the tablespace (SQLSTATE 428CB). The default is 4 096 byte (4K) pages. Any number of spaces is allowed between *integer* and K, including no space.

MANAGED BY SYSTEM
Specifies that the tablespace is to be a system managed space (SMS) tablespace.

system-containers
Specify the containers for an SMS tablespace.

USING *('container-string',...)*
For a SMS tablespace, identifies one or more containers that will belong to the tablespace and into which the tablespace's data will be stored. The *container-string* cannot exceed 240 bytes in length.

Each *container-string* can be an absolute or relative directory name. The directory name, if not absolute, is relative to the database directory. If any component of the directory name does not exist, it is created by the database manager. When a tablespace is dropped, all components created by the database manager are deleted. If the directory identified by container-string exist, it must not contain any files or subdirectories (SQLSTATE 428B2).

The format of *container-string* is dependent on the operating system. The containers are specified in the normal manner for the operating system. For example, a Windows directory path begins with a drive letter and a ":", while on UNIX-based systems, a path begins with a "/".

Note that remote resources (such as LAN-redirected drives or NFS-mounted file systems) are currently only supported when using Network Appliance Filers, IBM iSCSI, or IBM Network Attached Storage.

on-db-partitions-clause
Specifies the partition or partitions on which the containers are created in a partitioned database. If this clause is not specified, then the containers are created on the partitions in the database partition group that are not explicitly specified in any other *on-db-partitions-clause*s. For a SYSTEM TEMPORARY tablespace defined on database partition group IBMTEMPGROUP, when the *on-db-partitions-clause* is not specified, the containers will also be created on all new partitions added to the database.

MANAGED BY DATABASE
Specifies that the tablespace is to be a database managed space (DMS) tablespace.

database-containers
Specify the containers for a DMS tablespace.

USING
Introduces a container-clause.

container-clause
Specifies the containers for a DMS tablespace.

(FILE | DEVICE *'container-string' number-of-pages,...)*
For a DMS tablespace, identifies one or more containers that will belong to the tablespace and into which the tablespace's data will

be stored. The type of the container (either FILE or DEVICE) and its size (in PAGESIZE pages) are specified. The size can also be specified as an integer value followed by K (for kilobytes), M (for megabytes) or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages for the container. A mixture of FILE and DEVICE containers can be specified. The *container-string* cannot exceed 254 bytes in length.

For a FILE container, the *container-string* must be an absolute or relative file name. The file name, if not absolute, is relative to the database directory. If any component of the directory name does not exist, it is created by the database manager. If the file does not exist, it will be created and initialized to the specified size by the database manager. When a tablespace is dropped, all components created by the database manager are deleted.

Note: If the file exists it is overwritten and if it is smaller than specified it is extended. The file will not be truncated if it is larger than specified.

For a DEVICE container, the *container-string* must be a device name. The device must already exist.

All containers must be unique across all databases; a container can belong to only one tablespace. The size of the containers can differ, however optimal performance is achieved when all containers are the same size. The exact format of *container-string* is dependent on the operating system. The containers will be specified in the normal manner for the operating system.

Remote resources (such as LAN-redirected drives or NFS-mounted file systems) are currently only supported when using Network Appliance Filers, IBM iSCSI, or IBM Network Attached Storage.

on-db-partitions-clause
Specifies the partition or partitions on which the containers are created in a partitioned database. If this clause is not specified, then the containers are created on the partitions in the database partition group that are not explicitly specified in any other *on-db-partitions-clause*. For a SYSTEM TEMPORARY tablespace defined on database partition group IBMTEMPGROUP, when the *on-db-partitions-clause* is not specified, the containers will also be created on all new partitions added to the database.

on-db-partitions-clause
Specifies the partitions on which containers are created in a partitioned database.

ON DBPARTITIONNUMS
Keywords that indicate that specific partitions are specified. DBPARTITIONNUM is a synonym for DBPARTITIONNUMS.

db-partition-number1
Specify a database partition number.

TO *db-partition-number2*
Specify a range of partition numbers. The value of *db-partition-number2* must be greater than or equal to the value of *db-partition-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers are included in the partitions for which the containers are created if the partition is included in the database partition group of the tablespace.

The partition specified by number and every partition in the range of partitions must exist in the database partition group on which the tablespace is defined (SQLSTATE 42729). A partition-number may only appear explicitly or within a range in exactly one *on-db-partitions-clause* for the statement (SQLSTATE 42613).

EXTENTSIZE *number-of-pages*
Specifies the number of PAGESIZE pages that will be written to a container before skipping to the next container. The extent size value can also be specified as an integer value followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages value for extent size. The database manager cycles repeatedly through the containers as data is stored.

The default value is provided by the DFT_EXTENT_SZ configuration parameter.

PREFETCHSIZE *number-of-pages*
Specifies the number of PAGESIZE pages that will be read from the tablespace when data prefetching is being performed. The prefetch size value can also be specified as an integer value followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages value for prefetch size. Prefetching reads in data needed by a query prior to it being referenced by the query, so that the query need not wait for I/O to be performed.

The default value is provided by the DFT_PREFETCH_SZ configuration parameter.

BUFFERPOOL *bufferpool-name*
The name of the buffer pool used for tables in this tablespace. The buffer pool must exist (SQLSTATE 42704). If not specified, the default buffer pool (IBMDEFAULTBP) is used. The page size of the bufferpool must match the page size specified (or defaulted) for the tablespace (SQLSTATE 428CB). The database partition group of the tablespace must be defined for the bufferpool (SQLSTATE 42735).

OVERHEAD *number-of-milliseconds*
Any numeric literal (integer, decimal, or floating point) that specifies the I/O controller overhead and disk seek and latency time, in milliseconds. The number should be an average for all containers that belong to the tablespace, if not the same for all containers. This value is used to determine the cost of I/O during query optimization.

TRANSFERRATE *number-of-milliseconds*
Any numeric literal (integer, decimal, or floating point) that specifies the time to read one page into memory, in milliseconds. The number should be an average for all containers that belong to the tablespace, if not the same for all containers. This value is used to determine the cost of I/O during query optimization.

DROPPED TABLE RECOVERY
Dropped tables in the specified tablespace may be recovered using the RECOVER TABLE ON option of the ROLLFORWARD command. This clause can only be specified for a REGULAR tablespace (SQLSTATE 42613).

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NODE can be specified in place of for DBPARTITIONNUM
    - NODES can be specified in place of for DBPARTITIONNUMS
    - NODEGROUP can be specified in place of for DATABASE PARTITION GROUP
    - LONG can be specified in place of for LARGE
- Choosing between a database-managed space or a system-managed space for a tablespace is a fundamental choice involving trade-offs.
- When more than one TEMPORARY tablespace exists in the database, they will be used in round-robin fashion in order to balance their usage.
- In a partitioned database, if more than one database partition resides on the same physical node, the same device or specific path cannot be specified for such database partitions (SQLSTATE 42730). For this environment, either specify a unique *container-string* for each database partition or use a relative path name.

- You can specify a database partition expression for container string syntax when creating either SMS or DMS containers. You would typically specify the database partition expression if you were using multiple logical database partitions in the partitioned database system. This ensures that container names are unique across nodes (database partition servers). When you specify the expression, the database partition number is part of the container name or, if you specify additional arguments, the result of the argument is part of the container name.

  You use the argument " $N" ([blank]$N) to indicate a database partition expression. A database partition expression can be used anywhere in the container name, and multiple database partition expressions can be specified. Terminate the database partition expression with a space character; whatever follows the space is appended to the container name after the database partition expression is evaluated. If there is no space character in the container name after the database partition expression, it is assumed that the rest of the string is part of the expression. The argument can only be used in one of the following forms:

Table 8. Arguments for Creating Containers. Operators are evaluated from left to right. The database partition number in the examples is assumed to be 5.

Syntax	Example	Value
[blank]$N	" $N"	5
[blank]$N+[number]	" $N+1011"	1016
[blank]$N%[number]	" $N%3" [a]	2
[blank]$N+[number]%[number]	" $N+12%13"	4
[blank]$N%[number]+[number]	" $N%3+20"	22
[a] % is modulus.		

For example:

```
CREATE TABLESPACE TS1 MANAGED BY DATABASE USING
  (device '/dev/rcont $N' 20000)

On a two database partition system, the following containers
 would be created:
  /dev/rcont0  - on DATABASE PARTITION 0
  /dev/rcont1  - on DATABASE PARTITION 1


CREATE TABLESPACE TS2 MANAGED BY DATABASE USING
  (file '/DB2/containers/TS2/container $N+100' 10000)

On a four database partition system, the following containers
 would be created:
  /DB2/containers/TS2/container100  - on DATABASE PARTITION 0
  /DB2/containers/TS2/container101  - on DATABASE PARTITION 1
  /DB2/containers/TS2/container102  - on DATABASE PARTITION 2
  /DB2/containers/TS2/container103  - on DATABASE PARTITION 3
```

```
CREATE TABLESPACE TS3 MANAGED BY SYSTEM USING
  ('/TS3/cont $N%2','/TS3/cont $N%2+2')

On a two database partition system, the following containers
 would be created:
  /TS3/cont0  - On DATABASE PARTITION 0
  /TS3/cont2  - On DATABASE PARTITION 0
  /TS3/cont1  - On DATABASE PARTITION 1
  /TS3/cont3  - On DATABASE PARTITION 1


If database partition = 5, the containers:
  '/dbdir/node $N /cont1'
  '/ $N+1000 /file1'
  ' $N%10 /container'
  '/dir/ $N%5+2000 /dmscont'

are created as:
  '/dbdir/node5/cont1'
  '/1005/file1'
  '5/container'
  '/dir/2000/dmscont'
```

Examples:

Example 1: Create a regular DMS table space on a UNIX-based system using 3 devices of 10 000 4K pages each. Specify their I/O characteristics.

```
CREATE TABLESPACE PAYROLL
  MANAGED BY DATABASE
  USING (DEVICE'/dev/rhdisk6' 10000,
    DEVICE '/dev/rhdisk7' 10000,
    DEVICE '/dev/rhdisk8' 10000)
  OVERHEAD 24.1
  TRANSFERRATE 0.9
```

Example 2: Create a regular SMS table space on Windows NT/2000 using 3 directories on three separate drives, with a 64-page extent size, and a 32-page prefetch size.

```
CREATE TABLESPACE ACCOUNTING
  MANAGED BY SYSTEM
  USING ('d:\acc_tbsp', 'e:\acc_tbsp', 'f:\acc_tbsp')
  EXTENTSIZE 64
  PREFETCHSIZE 32
```

Example 3: Create a temporary DMS table space on Unix using 2 files of 50,000 pages each, and a 256-page extent size.

```
CREATE TEMPORARY TABLESPACE TEMPSPACE2
  MANAGED BY DATABASE
  USING (FILE '/tmp/tempspace2.f1' 50000,
    FILE '/tmp/tempspace2.f2' 50000)
  EXTENTSIZE 256
```

Example 4: Create a DMS table space on database partition group ODDNODEGROUP (partitions 1,3,5) on a Unix partitioned database. On all partitions, use the device /dev/rhdisk0 for 10 000 4K pages. Also specify a partition-specific device for each partition with 40 000 4K pages.

```
CREATE TABLESPACE PLANS
  MANAGED BY DATABASE
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn1hd01' 40000)
  ON DBPARTITIONNUM (1)
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn3hd03' 40000)
  ON DBPARTITIONNUM (3)
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn5hd05' 40000)
  ON DBPARTITIONNUM (5)
```

Related samples:

- "tbtemp.sqc -- How to use a declared temporary table (C)"
- "TbTemp.java -- How to use Declared Temporary Table (JDBC)"

CREATE TRANSFORM

The CREATE TRANSFORM statement defines transformation functions or methods, identified by a group name, that are used to exchange structured type values with host language programs and with external functions and methods.

Invocation:

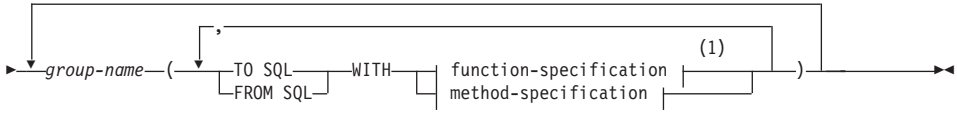
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

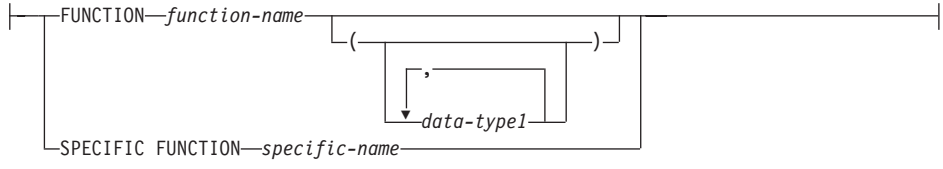
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- definer of the type identified by *type-name*, and EXECUTE privilege on every function specified.

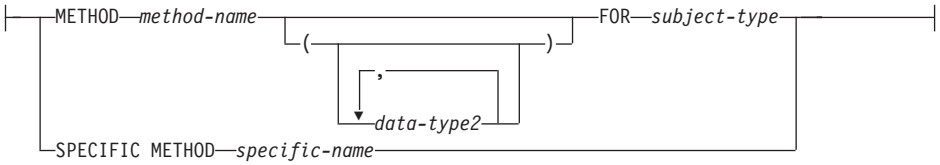
Syntax:



function-specification:



method-specification:



Notes:

1 The same clause must not be specified more than once.

Description:

TRANSFORM or **TRANSFORMS**
Indicates that one or more transform groups is being defined. Either version of the keyword can be specified.

FOR *type-name*
Specifies a name for the user-defined structured type for which the transform group is being defined.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified *type-name*. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for an unqualified *type-name*. The *type-name* must be the name of an existing user-defined type (SQLSTATE 42704), and it must be a structured type (SQLSTATE 42809). The structured type or any other structured type in the same type hierarchy must not have transforms already defined with the given group-name (SQLSTATE 42739).

group-name
Specifies the name of the transform group containing the TO SQL and FROM SQL functions or methods. The name does not need to be unique; but a transform group of this name (with the same TO SQL and/or FROM SQL direction defined) must not be previously defined for the specified *type-name* (SQLSTATE 42739). A *group-name* must be an SQL identifier, with a maximum of 18 characters in length (SQLSTATE 42622), and it may not include any qualifier prefix (SQLSTATE 42601). The *group-name* cannot begin with the prefix 'SYS', since this is reserved for database use (SQLSTATE 42939).

At most, one of each of the FROM SQL and TO SQL function designations may be specified for any given group (SQLSTATE 42628).

TO SQL
Defines the specific function used to transform a value to the SQL user-defined structured type format. The function must have all its parameters as built-in data types and the returned type is *type-name*.

FROM SQL
Defines the specific function used to transform a value to a built in data

type value representing the SQL user-defined structured type. The function must have one parameter of data type *type-name*, and return a built-in data type (or set of built-in data types).

WITH *function-specification*
There are several ways available to specify the function instance.

If FROM SQL is specified, *function-specification* must identify a function that meets the following requirements:

- There is one parameter of type *type-name*.
- The return type is a built-in type, or a row whose columns all have built-in types.
- The signature specifies either LANGUAGE SQL or the use of another FROM SQL transform function that has LANGUAGE SQL.

If TO SQL is specified, *function-specification* must identify a function that meets the following requirements:

- All parameters have built-in types.
- The return type is *type-name*.
- The signature specifies either LANGUAGE SQL or the use of another TO SQL transform function that has LANGUAGE SQL.

If *function-specification* identifies a function that does not meet these requirements (according to its use as a FROM SQL or a TO SQL transform function), an error is raised (SQLSTATE 428DC).

FUNCTION *function-name*
Identifies the particular function by name, and is valid only if there is exactly one function with the *function-name*. The identified function can have any number of parameters defined for it.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

If no function by this name exists in the named or implied schema, an error is raised (SQLSTATE 42704). If there is more than one specific instance of the function in the named or implied schema, an error is raised (SQLSTATE 42725). The standard function selection algorithm is not used.

FUNCTION *function-name* **(***data-tape1,...***)**
Provides the function signature, which uniquely identifies the function to be used. The standard function selection algorithm is not used.

function-name
Specifies the name of the function. In dynamic SQL statements,

the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(*data-type1,...***)**
The data types specified here must match the data types specified in the CREATE FUNCTION statement in the corresponding positions. Both the number of data types and the logical concatenation of the data types are used to identify the specific function.

If the data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead an empty set of parentheses can be coded to indicate that these attributes should be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE). However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(n) does not need to match the defined value for n, because 0<n<25 means REAL, and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE. Note that the FOR BIT DATA attribute is not considered part of the signature for matching purposes. For example, a CHAR FOR BIT DATA specified in the signature would match a function defined with CHAR only.

If no function with the specified signature exists in the named or implied schema, an error is raised (SQLSTATE 42883).

SPECIFIC FUNCTION *specific-name*
Identifies the particular user-defined function, using a specific name either specified or defaulted to at function creation time.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error is raised (SQLSTATE 42704).

WITH *method-specification*
There are several ways available to specify the method instance.

If FROM SQL is specified, *method-specification* must identify a method that meets the following requirements:

- The type specified by *subject-type* is of type *type-name*.
- The return type is a built-in type, or a row whose columns use built-in types.
- The signature specifies either LANGUAGE SQL or the use of another FROM SQL transform function or method that has LANGUAGE SQL.

If TO SQL is specified, *method-specification* must identify a method that meets the following requirements:

- The type specified by *subject-type* is of type *type-name*.
- All other parameter types, except the type of the implicit self parameter, are built-in types.
- The return type is *type-name*.
- The method is declared as SELF AS RESULT.
- The signature specifies either LANGUAGE SQL or the use of another TO SQL transform function or method that has LANGUAGE SQL.

If *method-specification* identifies a method that does not meet these requirements (according to its use as a FROM SQL or a TO SQL transform method), an error is raised (SQLSTATE 428DC).

METHOD *method-name* **FOR** *subject-type*
Identifies the particular method by name and type, and is valid only if there is exactly one method with the specified properties. The *method-name* identifier is an unqualified name. The identified method can have any number of parameters defined for it.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

If no method by this name exists for the specified type in the implied schema, an error is raised (SQLSTATE 42704). If there is more than one specific instance of the method in the implied schema, an error is raised (SQLSTATE 42725). The standard method resolution algorithm is not used.

METHOD *method-name* **(***data-tape2,...***) FOR** *subject-type*
Provides the method signature, which uniquely identifies the method to be used. The standard method resolution algorithm is not used.

method-name **FOR** *subject-type*
Specifies the name of the method. The *method-name* identifier is an unqualified name. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(*data-type2,...***)**
The data types specified here must match the data types specified in the method creation statement in the corresponding positions. Both the number of data types and the logical concatenation of the data types are used to identify the specific method.

If the data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead an empty set of parentheses can be coded to indicate that these attributes should be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE). However, if length, precision, or scale is coded, the value must exactly match that specified in the method creation statement.

A type of FLOAT(n) does not need to match the defined value for n, because 0<n<25 means REAL, and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE. Note that the FOR BIT DATA attribute is not considered part of the signature for matching purposes. For example, a CHAR FOR BIT DATA specified in the signature would match a method defined with CHAR only.

If no method with the specified signature exists in the implied schema, an error is raised (SQLSTATE 42883).

SPECIFIC METHOD *specific-name*
Identifies the particular user-defined method, using a specific name either specified or defaulted to at method creation time.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific method instance in the named or implied schema; otherwise, an error is raised (SQLSTATE 42704).

Rules:

- The one or more built-in types that are returned from the FROM SQL function or method should directly correspond to the one or more built-in types that are parameters of the TO SQL function or method. This is a logical consequence of the inverse relationship between these two functions. If this relationship between the FROM transform and the TO transform does not hold, an error is raised (SQLSTATE -3).

Notes:

- When a transform group is not specified in an application program (using the TRANSFORM GROUP precompile or bind option for static SQL, or the SET CURRENT DEFAULT TRANSFORM GROUP statement for dynamic SQL), the transform functions or methods in the transform group 'DB2_PROGRAM' are used (if defined) when the application program is retrieving or sending host variables that are based on the user-defined structured type identified by *type-name*. When retrieving a value of data type *type-name*, the FROM SQL transform is invoked to transform the structured type to the built-in data type returned by the transform function or method. Similarly, when sending a host variable that will be assigned to a value of data type *type-name*, the TO SQL transform is invoked to transform the built-in data type value to the structured type value. If the TO SQL transform is a method, an object of the appropriate dynamic type is created, and the TO SQL transform method is invoked with the newly created object as the subject argument. If a user-defined transform group is not specified, or a 'DB2_PROGRAM' group is not defined (for the given structured type), an error is raised (SQLSTATE 42741).
- The built-in data type representation for a structured type host variable must be assignable:
  - from the result of the FROM SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command (using retrieval assignment rules) and
  - to the parameter of the TO SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command (using storage assignment rules).

  If a host variable is not assignment compatible with the type required by the applicable transform function or method, an error is raised (for bind-in: SQLSTATE 42821; for bind-out: SQLSTATE 42806). For errors that result from string assignments, see “String Assignments”.
- The transform functions or methods identified in the default transform group named 'DB2_FUNCTION' are used whenever a user-defined function or method not written in SQL is invoked using the data type *type-name* as a parameter or returns type. This applies when the function or method does not specify the TRANSFORM GROUP clause. When invoking the function or method with an argument of data type *type-name*, the FROM SQL

transform is executed to transform the structured type to the built-in data type returned by the transform function or method. Similarly, when the returns data type of the function or method is of data type *type-name*, the TO SQL transform is invoked to transform the built-in data type value returned from the external function program into the structured type value.

If the TO SQL transform is a method, and the TO SQL transform is used to transform the result of a method invocation whose method has been declared as SELF AS RESULT, an object of the dynamic type of the subject is created. If the method is not declared as SELF AS RESULT, an object of the declared type of the result of the most specific dispatchable method is created. Otherwise, an object of the declared type of the result of the resolved function or method is created. The TO SQL transform method is invoked with the newly created object as the subject argument, together with the base type arguments passed back from the actual function or method invocation.

- If a structured type contains an attribute that is also a structured type, the associated transform functions or methods must recursively expand (or assemble) all nested structured types. This means that the results or parameters of the transform functions or methods consist only of the set of built-in types representing all base attributes of the subject structured type (including all its nested structured types). There is no ″cascading″ of transform functions or methods for handling nested structured types.
- The functions or methods identified in this statement are resolved according to the rules outlined above at the execution of this statement. When these functions are used (implicitly) in subsequent SQL statements, they do not undergo another resolution process. The transform functions or methods defined in this statement are recorded exactly as they are resolved in this statement. However, when a transform method is invoked, the algorithms for the REFER(Dynamic Dispatch of Methods) are applied.
- When attributes or subtypes of a given type are created or dropped, the transform functions for the user-defined structured type must also be changed.
- For a given transform group, the FROM SQL and TO SQL transforms can be specified in either the same *group-name* clause, in separate *group-name* clauses, or in separate CREATE TRANSFORM statements. The only restriction is that a given FROM SQL or TO SQL transform designation may not be redefined without first dropping the existing group definition. This allows you to define, for example, a FROM SQL transform for a given group first, and the corresponding TO SQL transform for the same group at a later time.

Examples:

Example 1: Create two transform groups that associate the user-defined structured type polygon with transform functions customized for C and Java, respectively.

```
CREATE TRANSFORM FOR POLYGON
  mystruct1 (FROM SQL WITH FUNCTION myxform_sqlstruct,
             TO SQL WITH FUNCTION myxform_structsql)
  myjava1   (FROM SQL WITH FUNCTION myxform_sqljava,
             TO SQL WITH FUNCTION myxform_javasql)
```

Example 2: Create two transform groups that associate the user-defined structured type polygon with transform methods customized for C and Java, respectively.

```
CREATE TRANSFORM FOR POLYGON
  mystruct1 (FROM SQL WITH METHOD myxform_sqlstruct FOR POLYGON,
             TO SQL WITH METHOD myxform_structsql FOR POLYGON)
  myjava1   (FROM SQL WITH METHOD myxform_sqljava FOR POLYGON,
             TO SQL WITH METHOD myxform_javasql FOR POLYGON)
```

Example 3: Create a transform group for a type that is not in the current schema.

```
CREATE TRANSFORM FOR NEWTON.POLYGON
  mystruct1 (FROM SQL WITH METHOD myxform_sqlstruct FOR NEWTON.POLYGON,
             TO SQL WITH METHOD myxform_structsql FOR NEWTON.POLYGON)
```

Related reference:

- "Assignments and comparisons" in the *SQL Reference, Volume 1*

CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger in the database.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement when the trigger is created must include at least one of the following:

- SYSADM or DBADM authority
- ALTER privilege on the table on which the BEFORE or AFTER trigger is defined
- CONTROL privilege on the view on which the INSTEAD OF TRIGGER is defined
- definer of the view on which the INSTEAD OF trigger is defined
- ALTERIN privilege on the schema of the table or view on which the trigger is defined

and one of:

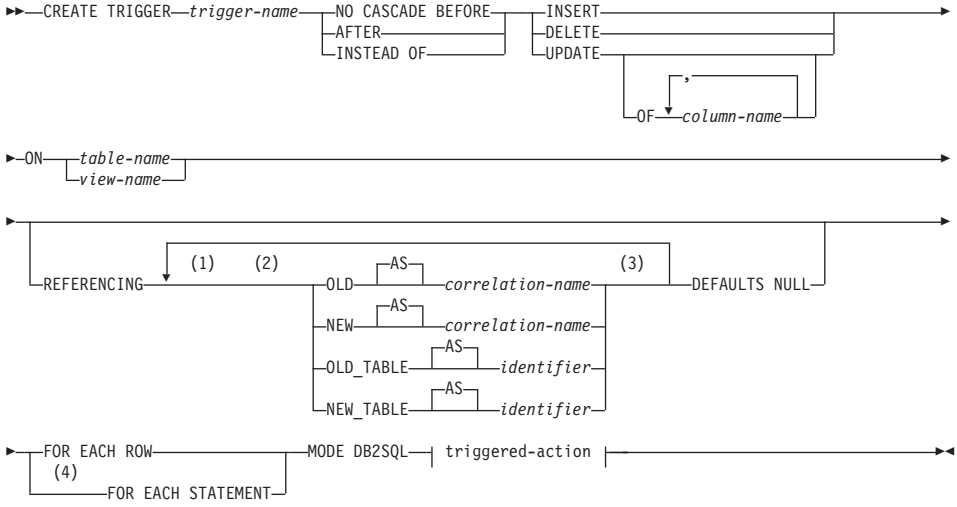
- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the trigger does not exist
- CREATEIN privilege on the schema, if the schema name of the trigger refers to an existing schema.

If the authorization ID of the statement does not have SYSADM or DBADM authority, the privileges that the authorization ID of the statement holds (without considering PUBLIC or group privileges) must include all of the following as long as the trigger exists:

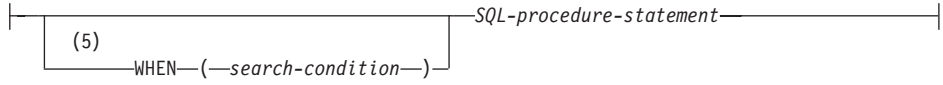
- SELECT privilege on the table on which the trigger is defined, if any transition variables or tables are specified
- SELECT privilege on any table or view referenced in the triggered action condition
- Necessary privileges to invoke the triggered SQL statements specified.

If a trigger definer can only create the trigger because the definer has SYSADM authority, then the definer is granted explicit DBADM authority for the purpose of creating the trigger.

Syntax:



triggered-action:



Notes:

1 OLD and NEW may only be specified once each.

2 OLD_TABLE and NEW_TABLE may only be specified once each, and only for AFTER triggers or INSTEAD OF triggers.

3 DEFAULTS NULL must be specified with a REFERENCING clause that specifies NEW or NEW_TABLE for INSTEAD OF triggers, and cannot be specified for any other trigger definition.

4 FOR EACH STATEMENT may not be specified for BEFORE triggers or INSTEAD OF triggers.

5 WHEN condition may not be specified for INSTEAD OF triggers.

Description:

trigger-name
Names the trigger. The name, including the implicit or explicit schema name must not identify a trigger already described in the catalog (SQLSTATE 42710). If a two part name is specified, the schema name cannot begin with ″SYS″ (SQLSTATE 42939).

NO CASCADE BEFORE
Specifies that the associated triggered action is to be applied before any changes caused by the actual update of the subject table are applied to the database. It also specifies that the triggered action of the trigger will not cause other triggers to be activated.

AFTER
Specifies that the associated triggered action is to be applied after the changes caused by the actual update of the subject table are applied to the database.

INSTEAD OF
Specifies that the associated triggered action replaces the action against the subject view. Only one INSTEAD OF trigger is allowed for each kind of operation on a given subject view (SQLSTATE 428FP).

INSERT
Specifies that the triggered action associated with the trigger is to be executed whenever an INSERT operation is applied to the subject table or subject view.

DELETE
Specifies that the triggered action associated with the trigger is to be executed whenever a DELETE operation is applied to the subject table or subject view.

UPDATE
Specifies that the triggered action associated with the trigger is to be executed whenever an UPDATE operation is applied to the subject table or subject view, subject to the columns specified or implied.

If the optional *column-name* list is not specified, every column of the table is implied. Therefore, omission of the *column-name* list implies that the trigger will be activated by the update of any column of the table.

OF *column-name*,...
Each *column-name* specified must be a column of the base table (SQLSTATE 42703). If the trigger is a BEFORE trigger, the *column-name* specified may not be a generated column other than the identity column (SQLSTATE 42989). No *column-name* shall appear more than once in the *column-name* list (SQLSTATE 42711). The trigger will only be activated by the update of a column identified in the *column-name* list. This clause cannot be specified for an INSTEAD OF trigger (SQLSTATE 42613).

ON

table-name
Designates the subject table of the BEFORE trigger or AFTER trigger definition. The name must specify a base table or an alias that

resolves to a base table (SQLSTATE 42704 or 42809). The name must not specify a catalog table (SQLSTATE 42832), a materialized query table (SQLSTATE 42997), a declared temporary table (SQLSTATE 42995), or a nickname (SQLSTATE 42809).

view-name
Designates the subject view of the INSTEAD OF trigger definition. The name must specify an untyped view or an alias that resolves to an untyped view (SQLSTATE 42704 or 42809). The name must not specify a catalog view (SQLSTATE 42832). The name must not specify a view that is defined using WITH CHECK OPTION (a symmetric view), or a view on which a symmetric view has been defined, directly or indirectly (SQLSTATE 428FQ).

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 names identify the complete set of affected rows. Each row affected by the triggering SQL operation is available to the triggered action by qualifying columns with *correlation-names* specified as follows.

OLD AS *correlation-name*
Specifies a correlation name which identifies the row state prior to the triggering SQL operation.

NEW AS *correlation-name*
Specifies a correlation name which identifies the row state as modified by the triggering SQL operation and by 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 a temporary table name specified as follows.

OLD_TABLE AS *identifier*
Specifies a temporary table name which identifies the set of affected rows prior to the triggering SQL operation.

NEW_TABLE AS *identifier*
Specifies a temporary table name which identifies the affected rows as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed.

The following rules apply to the REFERENCING clause:

- None of the OLD and NEW correlation names and the OLD_TABLE and NEW_TABLE names can be identical (SQLSTATE 42712).

- Only one OLD and one NEW *correlation-name* may be specified for a trigger (SQLSTATE 42613).
- Only one OLD_TABLE and one NEW_TABLE *identifier* may be specified for a trigger (SQLSTATE 42613).
- The OLD *correlation-name* and the OLD_TABLE *identifier* can only be used if the trigger event is either a DELETE operation or an UPDATE operation (SQLSTATE 42898). If the operation is a DELETE operation, OLD *correlation-name* captures the value of the deleted row. If it is an UPDATE operation, it captures the value of the row before the UPDATE operation. The same applies to the OLD_TABLE *identifier* and the set of affected rows.
- The NEW *correlation-name* and the NEW_TABLE *identifier* can only be used if the trigger event is either an INSERT operation or an UPDATE operation (SQLSTATE 42898). In both operations, the value of NEW captures the new state of the row as provided by the original operation and as modified by any BEFORE trigger that has executed to this point. The same applies to the NEW_TABLE *identifier* and the set of affected rows.
- OLD_TABLE and NEW_TABLE *identifier*s cannot be defined for a BEFORE trigger (SQLSTATE 42898).
- OLD and NEW *correlation-name*s cannot be defined for a FOR EACH STATEMENT trigger (SQLSTATE 42899).
- Transition tables cannot be modified (SQLSTATE 42807).
- The total of the references to the transition table columns and transition variables in the triggered-action cannot exceed the limit for the number of columns in a table or the sum of their lengths cannot exceed the maximum length of a row in a table (SQLSTATE 54040).
- The scope of each *correlation-name* and each *identifier* is the entire trigger definition.

FOR EACH ROW
Specifies that the triggered action is to be applied once for each row of the subject table or subject view that is affected by the triggering SQL operation.

FOR EACH STATEMENT
Specifies that the triggered action is to be applied only once for the whole statement. This type of trigger granularity cannot be specified for a BEFORE trigger or an INSTEAD OF trigger (SQLSTATE 42613). If specified, an UPDATE or DELETE trigger is activated, even if no rows are affected by the triggering UPDATE or DELETE statement.

MODE DB2SQL
This clause is used to specify the mode of triggers. This is the only valid mode currently supported.

triggered-action

Specifies the action to be performed when a trigger is activated. A triggered-action is composed of an *SQL-procedure-statement* and by an optional condition for the execution of the *SQL-procedure-statement*.

WHEN (*search-condition***)**

Specifies a condition that is true, false, or unknown. The *search-condition* provides a capability to determine whether or not a certain triggered action should be executed.

The associated action is performed only if the specified search condition evaluates as true. If the WHEN clause is omitted, the associated *SQL-procedure statement* is always performed.

The WHEN clause may not be specified for INSTEAD OF triggers (SQLSTATE 42613).

SQL-procedure-statement

The *SQL-procedure-statement* can contain a dynamic compound statement or any of the SQL control statements listed in "Compound SQL (Dynamic)".

If the trigger is a BEFORE trigger, an *SQL-procedure-statement* can also include one of the following:

- a fullselect (A common-table-expression may precede a fullselect.)
- a SET variable statement.

If the trigger is an AFTER trigger or an INSTEAD OF trigger, an *SQL-procedure-statement* can also include one of the following:

- an INSERT SQL statement (not using nicknames)
- a searched UPDATE SQL statement (not using nicknames)
- a searched DELETE SQL statement (not using nicknames)
- a SET variable statement
- a fullselect (A common-table-expression may precede a fullselect.)

The *SQL-procedure-statement* must not contain a statement that is not supported (SQLSTATE 42987).

The *SQL-procedure-statement* cannot reference an undefined transition variable (SQLSTATE 42703), a federated object (SQLSTATE 42997), or a declared temporary table (SQLSTATE 42995).

The *SQL-procedure-statement* in a BEFORE trigger cannot reference a materialized query table defined with REFRESH IMMEDIATE (SQLSTATE 42997).

The *SQL-procedure-statement* in a BEFORE trigger cannot reference a generated column, other than the identity column, in the new transition variable (SQLSTATE 42989).

Notes:

- Adding a trigger to a table that already has rows in it will not cause any triggered actions to be activated. Thus, if the trigger is designed to enforce constraints on the data in the table, those constraints may not be satisfied by the existing rows.
- If the events for two triggers occur simultaneously (for example, if they have the same event, activation time, and subject tables), then the first trigger created is the first to execute.
- 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 specified without an explicit column list, then an update to the new column will cause the activation of the trigger.
  - The column will not be visible in the triggered action of any previously defined trigger.
  - The OLD_TABLE and NEW_TABLE transition tables will not contain this column. Thus, the result of performing a ″SELECT *″ on a transition table will not contain the added column.
- If a column is added to any table referenced in a triggered action, the new column will not be visible to the triggered action.
- The result of a fullselect specified in the *SQL-procedure-statement* is not available inside or outside of the trigger.
- A before delete trigger defined on a table involved in a cycle of cascaded referential constraints should not include references to the table on which it is defined or any other table modified by cascading during the evaluation of the cycle of referential integrity constraints. The results of such a trigger are data dependent and therefore may not produce consistent results.

  In its simplest form, this means that a before delete trigger on a table with a self-referencing referential constraint and a delete rule of CASCADE should not include any references to the table in the *triggered-action*.
- The creation of a trigger causes certain packages to be marked invalid:
  - If an update trigger without an explicit column list is created, then packages with an update usage on the target table or view are invalidated.
  - If an update trigger with a column list is created, then packages with update usage on the target table are only invalidated if the package also has an update usage on at least one column in the *column-name* list of the CREATE TRIGGER statement.

  - If an insert trigger is created, packages that have an insert usage on the target table or view are invalidated.
  - If a delete trigger is created, packages that have a delete usage on the target table or view are invalidated.
- A package remains invalid until the application program is explicitly bound or rebound, or it is executed and the database manager automatically rebinds it.
- ***Inoperative triggers***: An *inoperative trigger* is a trigger that is no longer available and is therefore never activated. A trigger becomes inoperative if:
  - a privilege that the creator of the trigger is required to have for the trigger to execute is revoked
  - an object such as a table, view or alias, upon which the triggered action is dependent, is dropped
  - a view, upon which the triggered action is dependent, becomes inoperative
  - an alias that is the subject table of the trigger is dropped.

  In practical terms, an inoperative trigger is one in which a trigger definition has been dropped as a result of cascading rules for DROP or REVOKE statements. For example, when an view is dropped, any trigger with an *SQL-procedure-statement* defined using that view is made inoperative.

  When a trigger is made inoperative, all packages with statements performing operations that were activating the trigger will be marked invalid. When the package is rebound (explicitly or implicitly) the inoperative trigger is completely ignored. Similarly, applications with dynamic SQL statements performing operations that were activating the trigger will also completely ignore any inoperative triggers.

  The trigger name can still be specified in the DROP TRIGGER and COMMENT ON TRIGGER statements.

  An inoperative trigger may be recreated by issuing a CREATE TRIGGER statement using the definition text of the inoperative trigger. This trigger definition text is stored in the TEXT column of the SYSCAT.TRIGGERS catalog view. Note that there is no need to explicitly drop the inoperative trigger in order to recreate it. Issuing a CREATE TRIGGER statement with the same *trigger-name* as an inoperative trigger will cause that inoperative trigger to be replaced with a warning (SQLSTATE 01595).

  Inoperative triggers are indicated by an X in the VALID column of the SYSCAT.TRIGGERS catalog view.
- ***Errors executing triggers***: Errors that occur during the execution of triggered SQL statements are returned using SQLSTATE 09000 unless the

error is considered severe. If the error is severe, the severe error SQLSTATE is returned. The SQLERRMC field of the SQLCA for non-severe error will include the trigger name, SQLCODE, SQLSTATE and as many tokens as will fit from the tokens of the failure.

The *SQL-procedure-statement* could include a SIGNAL SQLSTATE statement or a RAISE_ERROR function. In both these cases, the SQLSTATE returned is the one specified in the SIGNAL SQLSTATE statement or the RAISE_ERROR condition.

- Creating a trigger with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- A value generated by the database manager for an identity column is generated before the execution of any BEFORE triggers. Therefore, the generated identity value is visible to BEFORE triggers.
- A value generated by the database manager for a generated by expression column is generated after the execution of all BEFORE triggers.Therefore, the value generated by the expression is not visible to BEFORE triggers.
- ***Triggers and typed tables***: A BEFORE or AFTER trigger can be attached to a typed table at any level of a table hierarchy. If an SQL statement activates multiple triggers, the triggers will be executed in their creation order, even if they are attached to different tables in the typed table hierarchy.

  When a trigger is activated, its transition variables (OLD, NEW, OLD_TABLE and NEW_TABLE) may contain rows of subtables. However, they will contain only columns defined on the table to which they are attached.

  Effects of INSERT, UPDATE, and DELETE statements:

  – Row triggers: When an SQL statement is used to INSERT, UPDATE, or DELETE a table row, it activates row-triggers attached to the most specific table containing the row, and all supertables of that table. This rule is always true, regardless of how the SQL statement accesses the table. For example, when issuing an UPDATE EMP command, some of the updated rows may be in the subtable MGR. For EMP rows, the row-triggers attached to EMP and its supertables are activated. For MGR rows, the row-triggers attached to MGR and its supertables are activated.
  – Statement triggers: An INSERT, UPDATE, or DELETE statement activates statement-triggers attached to tables (and their supertables) that could be affected by the statement. This rule is always true, regardless of whether any actual rows in these tables were affected. For example, on an INSERT INTO EMP command, statement-triggers for EMP and its supertables are activated. As another example, on either an UPDATE EMP or DELETE EMP command, statement triggers for EMP and its supertables and subtables are activated, even if no subtable rows were updated or deleted. Likewise, a UPDATE ONLY (EMP) or DELETE

ONLY (EMP) command will activate statement-triggers for EMP and its supertables, but not statement-triggers for subtables.

Effects of DROP TABLE statements: A DROP TABLE statement does not activate any triggers that are attached to the table being dropped. However, if the dropped table is a subtable, all the rows of the dropped table are considered to be deleted from its supertables. Therefore, for a table T:

- Row triggers: DROP TABLE T activates row-type delete-triggers that are attached to all supertables of T, for each row of T.
- Statement triggers: DROP TABLE T activates statement-type delete-triggers that are attached to all supertables of T, regardless of whether T contains any rows.

Actions on Views: To predict what triggers are activated by an action on a view, use the view definition to translate that action into an action on base tables. For example:

1. An SQL statement performs UPDATE V1, where V1 is a typed view with a subview V2. Suppose V1 has underlying table T1, and V2 has underlying table T2. The statement could potentially affect rows in T1, T2, and their subtables, so statement triggers are activated for T1 and T2 and all their subtables and supertables.
2. An SQL statement performs UPDATE V1, where V1 is a typed view with a subview V2. Suppose V1 is defined as SELECT ... FROM ONLY(T1) and V2 is defined as SELECT ... FROM ONLY(T2). Since the statement cannot affect rows in subtables of T1 and T2, statement triggers are activated for T1 and T2 and their supertables, but not their subtables.
3. An SQL statement performs UPDATE ONLY(V1), where V1 is a typed view defined as SELECT ... FROM T1. The statement can potentially affect T1 and its subtables. Therefore, statement triggers are activated for T1 and all its subtables and supertables.
4. An SQL statement performs UPDATE ONLY(V1), where V1 is a typed view defined as SELECT ... FROM ONLY(T1). In this case, T1 is the only table that can be affected by the statement, even if V1 has subviews and T1 has subtables. Therefore, statement triggers are activated only for T1 and its supertables.

Examples:

Example 1: Create two triggers that will result in the automatic tracking of the number of employees a company manages. The triggers will interact with the following tables:

EMPLOYEE table with these columns: ID, NAME, ADDRESS, and POSITION.

COMPANY_STATS table with these columns: NBEMP, NBPRODUCT, and REVENUE.

The first trigger 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:

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

The second trigger decrements the number of employees each time an employee leaves the company; that is, each time a row is deleted from the table EMPLOYEE:

```
CREATE TRIGGER FORMER_EMP
  AFTER DELETE ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1
```

Example 2: Create a trigger that ensures that whenever a parts record is updated, the following check and (if necessary) action is taken:

If the on-hand quantity is less than 10% of the maximum stocked quantity, then issue a shipping request ordering the number of items for the affected part to be equal to the maximum stocked quantity minus the on-hand quantity.

The trigger will interact with the PARTS table with these columns: PARTNO, DESCRIPTION, ON_HAND, MAX_STOCKED, and PRICE.

ISSUE_SHIP_REQUEST is a user-defined function that sends an order form for additional parts to the appropriate company.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (N.ON_HAND < 0.10 * N.MAX_STOCKED)
  BEGIN ATOMIC
  VALUES(ISSUE_SHIP_REQUEST(N.MAX_STOCKED - N.ON_HAND, N.PARTNO));
  END
```

Example 3: Create a trigger that will cause an error when an update occurs that would result in a salary increase greater than ten percent of the current salary.

```
CREATE TRIGGER RAISE_LIMIT
  AFTER UPDATE OF SALARY ON EMPLOYEE
  REFERENCING NEW AS N OLD AS O
  FOR EACH ROW MODE DB2SQL
  WHEN (N.SALARY > 1.1 * O.SALARY)
         SIGNAL SQLSTATE '75000' SET MESSAGE_TEXT='Salary increase>10%'
```

Example 4: Consider an application which records and tracks changes to stock prices. The database contains two tables, CURRENTQUOTE and QUOTEHISTORY.

```
Tables: CURRENTQUOTE (SYMBOL, QUOTE, STATUS)
        QUOTEHISTORY (SYMBOL, QUOTE, QUOTE_TIMESTAMP)
```

When the QUOTE column of CURRENTQUOTE is updated, the new quote should be copied, with a timestamp, to the QUOTEHISTORY table. Also, the STATUS column of CURRENTQUOTE should be updated to reflect whether the stock is:

1. rising in value;
2. at a new high for the year;
3. dropping in value;
4. at a new low for the year;
5. steady in value.

CREATE TRIGGER statements that accomplish this are as follows.

- Trigger Definition to set the status:

```
CREATE TRIGGER STOCK_STATUS
  NO CASCADE BEFORE UPDATE OF QUOTE ON CURRENTQUOTE
  REFERENCING NEW AS NEWQUOTE OLD AS OLDQUOTE
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
     SET NEWQUOTE.STATUS =
       CASE
          WHEN NEWQUOTE.QUOTE >
                (SELECT MAX(QUOTE) FROM QUOTEHISTORY
                WHERE SYMBOL = NEWQUOTE.SYMBOL
                AND YEAR(QUOTE_TIMESTAMP) = YEAR(CURRENT DATE) )
             THEN 'High'
          WHEN NEWQUOTE.QUOTE <
(SELECT MIN(QUOTE) FROM QUOTEHISTORY
                WHERE SYMBOL = NEWQUOTE.SYMBOL
                AND YEAR(QUOTE_TIMESTAMP) = YEAR(CURRENT DATE) )
             THEN 'Low'
          WHEN NEWQUOTE.QUOTE > OLDQUOTE.QUOTE
             THEN 'Rising'
          WHEN NEWQUOTE.QUOTE < OLDQUOTE.QUOTE
             THEN 'Dropping'
          WHEN NEWQUOTE.QUOTE = OLDQUOTE.QUOTE
             THEN 'Steady'
       END;
  END
```

- Trigger Definition to record change in QUOTEHISTORY table:

```
CREATE TRIGGER RECORD_HISTORY
  AFTER UPDATE OF QUOTE ON CURRENTQUOTE
  REFERENCING NEW AS NEWQUOTE
  FOR EACH ROW MODE DB2SQL
```

```
BEGIN ATOMIC
  INSERT INTO QUOTEHISTORY
    VALUES (NEWQUOTE.SYMBOL, NEWQUOTE.QUOTE, CURRENT TIMESTAMP);
END
```

Related reference:

- "Compound SQL (Dynamic)" on page 123

Related samples:

- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"
- "tbtrig.sqc -- How to use a trigger on a table (C)"
- "tbtrig.sqC -- How to use a trigger on a table (C++)"
- "trigsql.sqb -- How to use a trigger on a table (MF COBOL)"
- "TbTrig.java -- How to use triggers (JDBC)"
- "TbTrig.sqlj -- How to use triggers (SQLj)"

CREATE TYPE (Structured)

The CREATE TYPE statement defines a user-defined structured type. A user-defined structured type may include zero or more attributes. A structured type may be a subtype allowing attributes to be inherited from a supertype. Successful execution of the statement generates methods, for retrieving and updating values of attributes. Successful execution of the statement also generates functions, for constructing instances of a structured type used in a column, for casting between the reference type and its representation type, and for supporting the comparison operators (=, <>, <, <=, >, and >=) on the reference type.

The CREATE TYPE statement also defines any method specifications for user-defined methods to be used with the user-defined structured type.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

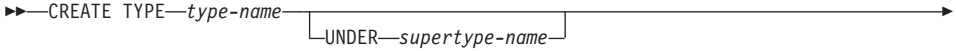
Authorization:

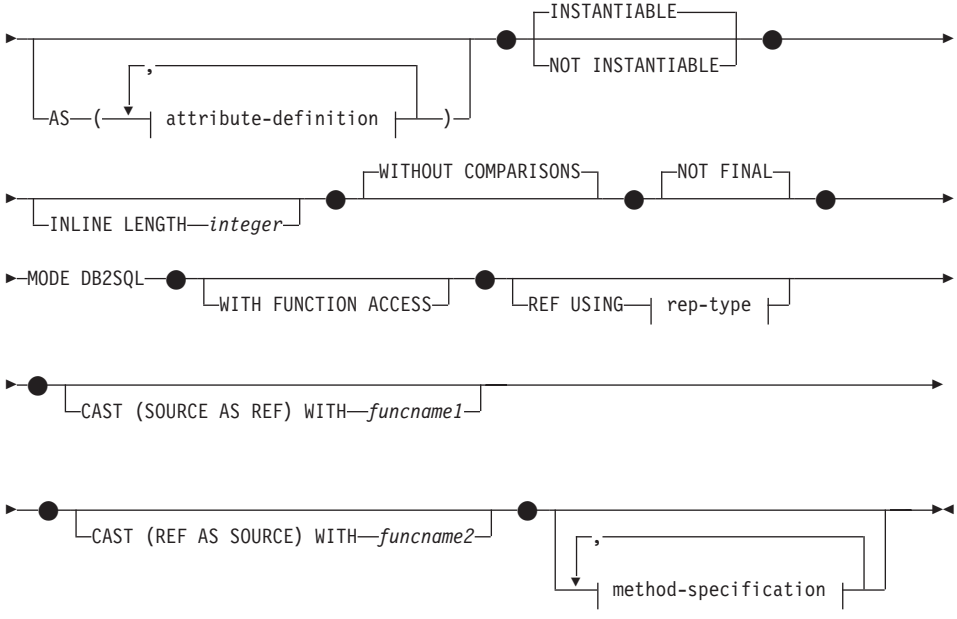
The privileges held by the authorization ID of the statement must include as least one of the following:

- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the database, if the schema name of the type does not refer to an existing schema.
- CREATEIN privilege on the schema, if the schema name of the type refers to an existing schema.

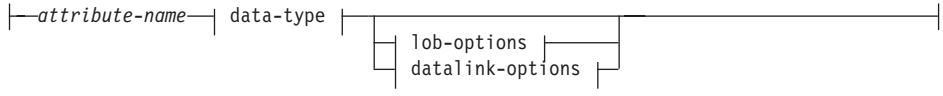
If UNDER is specified and the authorization ID of the statement is not the same as the definer of the root type of the type hierarchy, then SYSADM or DBADM authority is required.

Syntax:

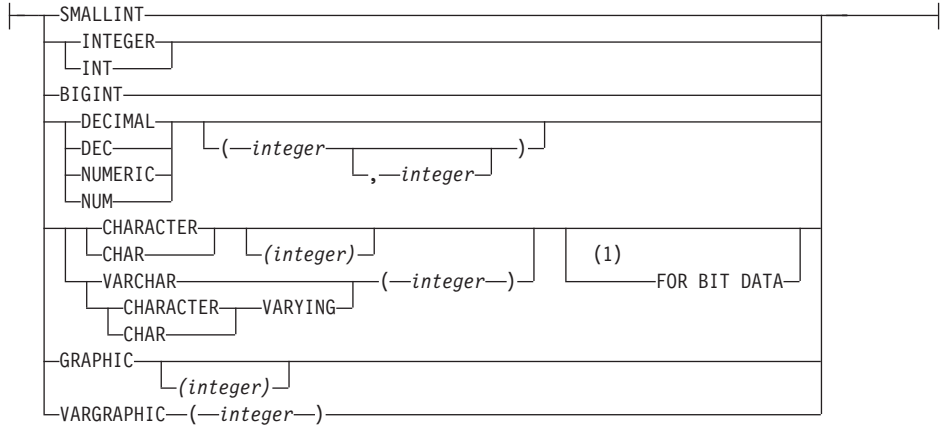




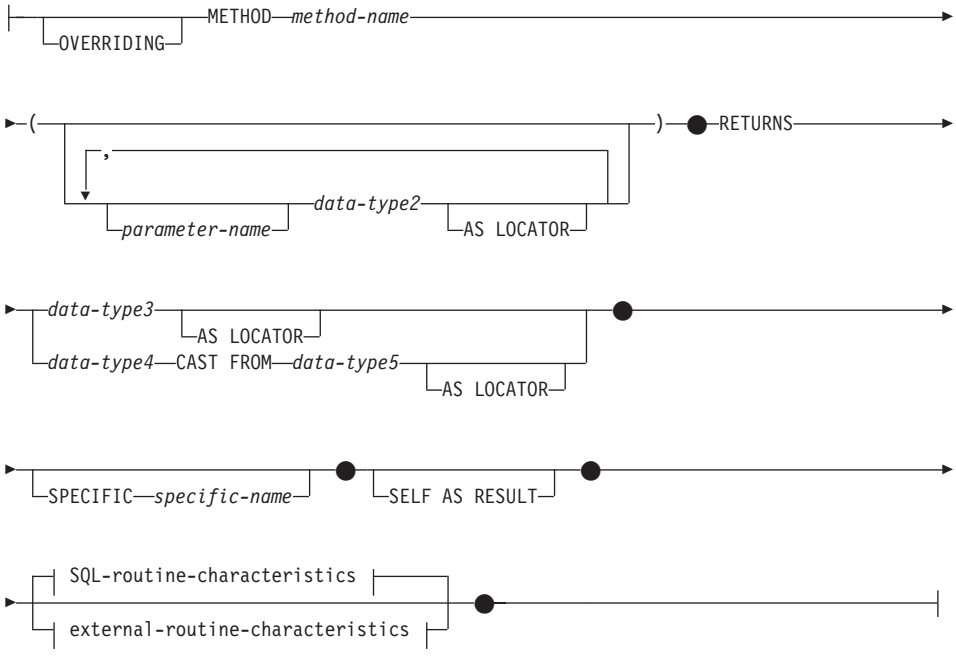
attribute-definition:



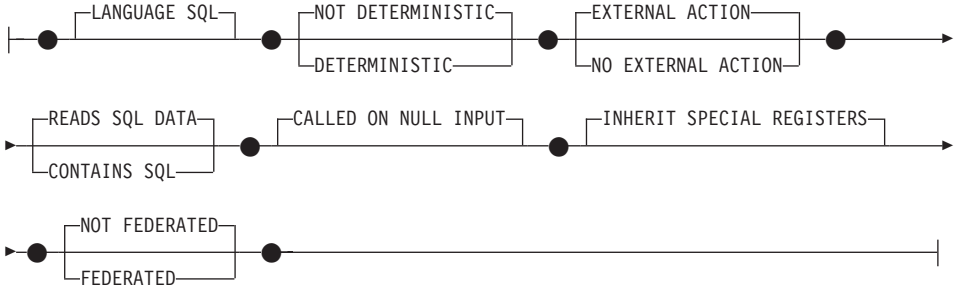
rep-type:



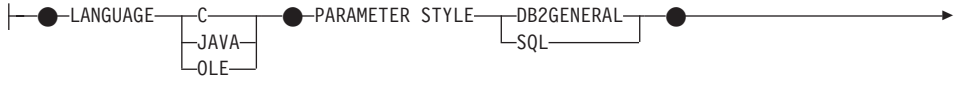
method-specification:

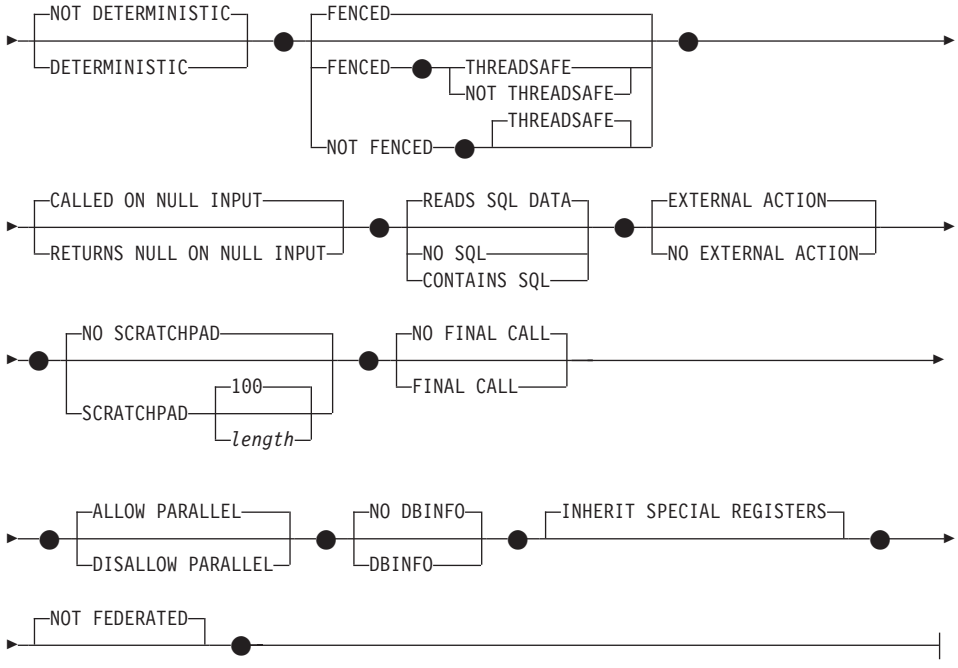


SQL-routine-characteristics:



external-routine-characteristics:





Notes:

1 The FOR BIT DATA clause may be specified in random order with the other column constraints that follow.

Description:

type-name

Names the type. The name, including the implicit or explicit qualifier, must not identify any other type (built-in, structured, or distinct) already described in the catalog. The unqualified name must not be the same as the name of a built-in data type or BOOLEAN (SQLSTATE 42918). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

The schema name (implicit or explicit) must not be greater than 8 bytes (SQLSTATE 42622).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *type-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

If a two-part *type-name* is specified, the schema name cannot begin with "SYS"; otherwise, an error (SQLSTATE 42939) is raised.

UNDER *supertype-name*
Specifies that this structured type is a subtype under the specified *supertype-name*. The *supertype-name* must identify an existing structured type (SQLSTATE 42704). If *supertype-name* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The structured type includes all the attributes of the supertype followed by the additional attributes given in the *attribute-definition*.

attribute-definition
Defines the attributes of the structured type.

attribute-name
The name of an attribute. The *attribute-name* cannot be the same as any other attribute of this structured type or any supertype of this structured type (SQLSTATE 42711).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as an *attribute-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

data-type
The data type of the attribute. It is one of the data types listed under "CREATE TABLE" other than LONG VARCHAR, LONG VARGRAPHIC, or a distinct type based on LONG VARCHAR or LONG VARGRAPHIC (SQLSTATE 42601). The data type must identify an existing data type (SQLSTATE 42704). If *data-type* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The description of various data types is given in "CREATE TABLE". If the attribute data type is a reference type, the target type of the reference must be a structured type that exists, or is created by this statement (SQLSTATE 42704).

A structured type defined with an attribute of type DATALINK can only be effectively used as the data type for a typed table or typed view (SQLSTATE 01641).

To prevent type definitions that would, at run time, permit an instance of the type to directly or indirectly contain another instance of the same type or one of its subtypes, a type cannot be defined such that one of its attribute types directly or indirectly uses itself (SQLSTATE 428EP).

lob-options
Specifies the options associated with LOB types (or distinct types based on LOB types). For a detailed description of *lob-options*, see "CREATE TABLE".

datalink-options
Specifies the options associated with DATALINK types (or distinct types based on DATALINK types). For a detailed description of *datalink-options*, see "CREATE TABLE".

Note that if no options are specified for a DATALINK type or distinct type sourced on DATALINK, LINKTYPE URL and NO LINK CONTROL options are the defaults.

INSTANTIABLE or **NOT INSTANTIABLE**
Determines whether an instance of the structured type can be created. Implications of not instantiable structured types are:

- no constructor function is generated for a non-instantiable type
- a non-instantiable type cannot be used as the type of a table or view (SQLSTATE 428DP)
- a non-instantiable type can be used as the type of a column (only null values or instances of instantiable subtypes can be inserted into the column.

To create instances of a non-instantiable type, instantiable subtypes must be created. If NOT INSTANTIABLE is specified, no instance of the new type can be created.

INLINE LENGTH *integer*
This option indicates the maximum size (in bytes) of a structured type column instance to store inline with the rest of the values in the row of a table. Instances of a structured type or its subtypes, that are larger than the specified inline length, are stored separately from the base table row, similar to the way that LOB values are handled.

If the specified INLINE LENGTH is smaller than the size of the result of the constructor function for the newly-created type (32 bytes plus 10 bytes per attribute) and smaller than 292 bytes, an error results (SQLSTATE 429B2). Note that the number of attributes includes all attributes inherited from the supertype of the type.

The INLINE LENGTH for the type, whether specified or a default value, is the default inline length for columns that use the structured type. This default can be overridden at CREATE TABLE time.

INLINE LENGTH has no meaning when the structured type is used as the type of a typed table.

The default INLINE LENGTH for a structured type is calculated by the system. In the formula given below, the following terms are used:

short attribute
: refers to an attribute with any of the following data types: SMALLINT, INTEGER, BIGINT, REAL, DOUBLE, FLOAT, DATE, or TIME. Also included are distinct types or reference types based on these types.

non-short attribute
: refers to an attribute of any of the remaining data types, or distinct types based on those data types.

The system calculates the default inline length as follows:

1. Determine the added space requirements for non-short attributes using the following formula:

   *space_for_non_short_attribute*s = SUM(*attributelength* + n)

   n is defined as:

   - 0 bytes for nested structured type attributes
   - 2 bytes for non-LOB attributes
   - 9 bytes for LOB attributes

   attributelength is based on the data type specified for the attribute as shown in Table 9.
2. Calculate the total default inline length using the following formula:

   default_length(*structured_type*) = (*number_of_attributes* * 10) + 32 + *space_for_non-short_attributes*

   number_of_attributes is the total number of attributes for the structured type, including attributes that are inherited from its supertype. However, *number_of_attributes* does not include any attributes defined for any subtype of *structured_type*.

Table 9. Byte Counts for Attribute Data Types

Attribute Data Type	Byte Count
DECIMAL	The integral part of (p/2)+1, where p is the precision
CHAR(n)	n
VARCHAR(n)	n
GRAPHIC(n)	n * 2
VARGRAPHIC(n)	n * 2
TIMESTAMP	10
DATALINK(n)	n + 54

Table 9. Byte Counts for Attribute Data Types (continued)

LOB Type	Each LOB attribute has a LOB descriptor in the structured type instance that points to the location of the actual value. The size of the descriptor varies according to the maximum length defined for the LOB attribute	
	Maximum LOB Length	LOB Descriptor Size
	1 024	72
	8 192	96
	65 536	120
	524 000	144
	4 190 000	168
	134 000 000	200
	536 000 000	224
	1 070 000 000	256
	1 470 000 000	280
	2 147 483 647	316
Distinct Type	Length of the source type of the distinct type	
Reference Type	Length of the built-in data type on which the reference type is based.	
Structured Type	inline_length(*attribute_type*)	

WITHOUT COMPARISONS
Indicates that there are no comparison functions supported for instances of the structured type.

NOT FINAL
Indicates that the structured type may be used as a supertype.

MODE DB2SQL
This clause is required and allows for direct invocation of the constructor function on this type.

WITH FUNCTION ACCESS
Indicates that all methods of this type and its subtypes, including methods created in the future, can be accessed using functional notation. This clause can be specified only for the root type of a structured type hierarchy (the UNDER clause is not specified) (SQLSTATE 42613). This clause is provided to allow the use of functional notation for those applications that prefer this form of notation over method invocation notation.

REF USING *rep-type*
Defines the built-in data type used as the representation (underlying data type) for the reference type of this structured type and all its subtypes. This clause can only be specified for the root type of a structured type

hierarchy (UNDER clause is not specified) (SQLSTATE 42613). The *rep-type* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, DATALINK, or structured type, and must have a length less than or equal to 32 672 bytes (SQLSTATE 42613).

If this clause is not specified for the root type of a structured type hierarchy, then REF USING VARCHAR(16) FOR BIT DATA is assumed.

CAST (SOURCE AS REF) WITH *funcname1*
Defines the name of the system-generated function that casts a value with the data type *rep-type* to the reference type of this structured type. A schema name must not be specified as part of *funcname1* (SQLSTATE 42601). The cast function is created in the same schema as the structured type. If the clause is not specified, the default value for *funcname1* is *type-name* (the name of the structured type). A function signature matching *funcname1(rep-type)* must not already exist in the same schema (SQLSTATE 42710).

CAST (REF AS SOURCE) WITH *funcname2*
Defines the name of the system-generated function that casts a reference type value for this structured type to the data type *rep-type*. A schema name must not be specified as part of *funcname2* (SQLSTATE 42601). The cast function is created in the same schema as the structured type. If the clause is not specified, the default value for *funcname2* is *rep-type* (the name of the representation type).

method-specification
Defines the methods for this type. A method cannot actually be used until it is given a body with a CREATE METHOD statement (SQLSTATE 42884).

OVERRIDING
Specifies that the method being defined overrides a method of a supertype of the type being defined. Overriding enables one to re-implement methods in subtypes, thereby providing more specific functionality. Overriding is not supported for the following types of methods:

- Table and row methods
- External methods declared with PARAMETER STYLE JAVA
- Methods that can be used as predicates in an index extension
- System-generated mutator or observer methods

Attempting to override such a method will result in an error (SQLSTATE 42745).

If a method is to be a valid overriding method, there must already exist one original method for one of the proper supertypes of the type

being defined, and the following relationships must exist between the overriding method and the original method:

- The method name of the method being defined and the original method are equivalent.
- The method being defined and the original method have the same number of parameters.
- The data type of each parameter of the method being defined and the data type of the corresponding parameters of the original method are identical. This requirement excludes the implicit SELF parameter.

If such an original method does not exist, an error is returned (SQLSTATE 428FV).

The overriding method inherits the following attributes from the original method:

- Language
- Determinism indication
- External action indication
- An indication whether this method should be called if any of its arguments is the null value
- Result cast (if specified in the original method)
- SELF AS RESULT indication
- The SQL-data access or CONTAINS SQL indication
- For external methods:
  - Parameter style
  - Locator indication of the parameters and of the result (if specified in the original method)
  - FENCED, SCRATCHPAD, FINAL CALL, ALLOW PARALLEL, and DBINFO indication
  - FEDERATED, INHERIT SPECIAL REGISTER, and THREADSAFE indication

method-name
Names the method being defined. It must be an unqualified SQL identifier (SQLSTATE 42601). The method name is implicitly qualified with the schema used for CREATE TYPE.

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *method-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

In general, the same name can be used for more than one method if there is some difference in their signatures.

parameter-name
Identifies the parameter name. It cannot be SELF, which is the name for the implicit subject parameter of a method (SQLSTATE 42734). If the method is an SQL method, all its parameters must have names (SQLSTATE 42629). If the method being declared overrides another method, the parameter name must be exactly the same as the name of the corresponding parameter of the overridden method; otherwise, an error is returned (SQLSTATE 428FV).

data-type2
Specifies the data type of each parameter. One entry in the list must be specified for each parameter that the method will expect to receive. No more than 90 parameters are allowed, including the implicit SELF parameter. If this limit is exceeded, an error is raised (SQLSTATE 54023).

SQL data type specifications and abbreviations which may be specified as a column-type in a CREATE TABLE statement and have a correspondence in the language that is being used to write the method may be specified. Refer to the language-specific sections of the Application Development Guide for details on the mapping between SQL data types and host language data types with respect to user-defined functions and methods.

Note: If the SQL data type in question is a structured type, there is no default mapping to a host language data type. A user-defined transform function must be used to create a mapping between the structured type and the host language data type.

DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815).

REF may be specified, but it does not have a defined scope. Inside the body of the method, a reference-type can be used in a path-expression only by first casting it to have a scope. Similarly, a reference returned by a method can be used in a path-expression only by first casting it to have a scope.

AS LOCATOR
For LOB types or distinct types which are based on a LOB type, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed to the method instead of the actual value. This saves greatly in the number of bytes passed to the method,

and may save as well in performance, particularly in the case where only a few bytes of the value are actually of interest to the method.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

If the method being declared overrides another method, the AS LOCATOR indication of the parameter must match exactly the AS LOCATOR indication of the corresponding parameter of the overridden method (SQLSTATE 428FV).

If the method being declared overrides another method, the FOR BIT DATA indication of each parameter must match exactly the FOR BIT DATA indication of the corresponding parameter of the overridden method. (SQLSTATE 428FV).

RETURNS

This mandatory clause identifies the method's result.

data-type3

Specifies the data type of the method's result. In this case, exactly the same considerations apply as for the parameters of methods described above under data-type2.

AS LOCATOR

For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the method instead of the actual value.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

If the method overrides another method, *data-type3* must be a subtype of the data type of the result of the overridden method if this data type is a structured type; otherwise both data types must be identical (SQLSTATE 428FV).

data-type4 **CAST FROM** *data-type5*

Specifies the data type of the method's result.

This clause is used to return a different data type to the invoking statement from the data type returned by the method code. The *data-type5* must be castable to the *data-type4* parameter. If it is not castable, an error is raised (SQLSTATE 42880).

Since the length, precision or scale for *data-type4* can be inferred from *data-type5*, it not necessary (but still permitted) to specify the length, precision, or scale for parameterized types specified for *data-type4*. Instead, empty parentheses may be used (VARCHAR(), for example). FLOAT() cannot be used (SQLSTATE 42601), since the parameter value indicates different data types (REAL or DOUBLE).

A distinct type is not valid as the type specified in data-type5 (SQLSTATE 42815).

The cast operation is also subject to runtime checks that might result in conversion errors being raised.

AS LOCATOR
For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the method instead of the actual value.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

If the method being defined overrides another method, the FOR BIT DATA clause cannot be specified (SQLSTATE 428FV).

SPECIFIC *specific-name*
Provides a unique name for the instance of the method that is being defined. This specific name can be used when creating the method body or dropping the method. It can never be used to invoke the method. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a schema-name followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another specific method name that exists at the application server; otherwise an error is raised (SQLSTATE 42710).

The *specific-name* may be the same as an existing *method-name*.

If no qualifier is specified, the qualifier that was used for *type-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *type-name* or an error is raised (SQLSTATE 42882).

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

SELF AS RESULT
Identifies this method as a type-preserving method, which means the following:

- The declared return type must be the same as the declared subject-type (SQLSTATE 428EQ).
- When an SQL statement is compiled and resolves to a type preserving method, the static type of the result of the method is the same as the static type of the subject argument.
- The method must be implemented in such a way that the dynamic type of the result is the same as the dynamic type of the subject argument (SQLSTATE 2200G), and the result cannot be NULL (SQLSTATE 22004).

If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

SQL-routine-characteristics
Specifies the characteristics of the method body that will be defined for this type using CREATE METHOD.

LANGUAGE SQL
This clause is used to indicate that the method is written in SQL with a single RETURN statement. The method body is specified using the CREATE METHOD statement.

NOT DETERMINISTIC or **DETERMINISTIC**
This optional clause specifies whether the method always returns the same results for given argument values (DETERMINISTIC) or whether the method depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC method must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. NOT DETERMINISTIC must be explicitly or implicitly specified if the body of the method accesses a special register, or calls another non-deterministic routine (SQLSTATE 428C2).

EXTERNAL ACTION or **NO EXTERNAL ACTION**
This optional clause specifies whether or not the method takes some action that changes the state of an object not managed by the database manager. Optimizations that assume methods have no external

impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

READS SQL DATA or **CONTAINS SQL**
Indicates what type of SQL statements can be executed. Because the SQL statement supported is the RETURN statement, the distinction has to do with whether or not the expression is a subquery.

READS SQL DATA
Indicates that SQL statements that do not modify SQL data can be executed by the method (SQLSTATE 42985). Nicknames cannot be referenced in the SQL statement (SQLSTATE 42997).

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the method (SQLSTATE 42985).

CALLED ON NULL INPUT
This optional clause indicates that regardless of whether any arguments are null, the user-defined method is called. It can return a null value or a normal (non-null) value. However, responsibility for testing for null argument values lies with the method.

If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

NULL CALL can be used as a synonym for CALLED ON NULL INPUT.

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the method will inherit their initial values from the environment of the invoking statement. For a method invoked in the select-statement of a cursor, the initial values are inherited from the environment in which the cursor is opened. For a routine invoked in a nested object (for example a trigger or view), the initial values are inherited from the run-time environment (not inherited from the object definition).

No changes to the special registers are passed back to the invoker of the function.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

FEDERATED or NOT FEDERATED
This optional clause specifies whether federated objects (federated routines, federated views, nicknames or OLE DB functions) can be used in the method body. If FEDERATED is specified, federated objects can be accessed from SQL statements in the method. NOT

FEDERATED is the default. Statements within the method that access federated objects may be subject to special authorization rules. If NOT FEDERATED is specified, federated objects cannot be used in any SQL statement in the method (SQLSTATE 429BA).

external-routine-characteristics

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the user-defined method body is written.

C This means the database manager will call the user-defined method as if it were a C function. The user-defined method must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA
This means the database manager will call the user-defined method as a method in a Java class.

OLE
This means the database manager will call the user-defined method as if it were a method exposed by an OLE automation object. The method must conform with the OLE automation data types and invocation mechanism as described in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is only supported for user-defined methods stored in Windows 32-bit operating systems. THREADSAFE may not be specified for methods defined with LANGUAGE OLE (SQLSTATE 42613).

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from methods.

DB2GENERAL
Used to specify the conventions for passing parameters to and returning the value from external methods that are defined as a method in a Java class. This can only be specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

SQL
Used to specify the conventions for passing parameters to and returning the value from external methods that conform to C language calling and linkage conventions or methods exposed by OLE automation objects. This must be specified when either LANGUAGE C or LANGUAGE OLE is used.

DETERMINISTIC or **NOT DETERMINISTIC**
This optional clause specifies whether the method always returns the same results for given argument values (DETERMINISTIC) or whether the method depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC method must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC.

An example of a NOT DETERMINISTIC method would be a method that randomly returns a serial number of an employee in a department. An example of a DETERMINISTIC method would be a method that calculates the area of a polygon.

FENCED or **NOT FENCED**
This clause specifies whether the method is considered ″safe″ to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a method is registered as FENCED, the database manager protects its internal resources (data buffers, for example) from access by the method. Most methods will have the option of running as FENCED or NOT FENCED. In general, a method running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION:
Use of NOT FENCED for methods not adequately checked out can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user-defined methods are used.

Only FENCED can be specified for a method with LANGUAGE OLE or NOT THREADSAFE (SQLSTATE 42613).

If the method is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

Either SYSADM authority, DBADM authority, or a special authority (CREATE_NOT_FENCED_ROUTINE) is required to register a method as NOT FENCED.

THREADSAFE or **NOT THREADSAFE**
Specifies whether the method is considered "safe" to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the method is defined with LANGUAGE other than OLE:

- If the method is defined as THREADSAFE, the database manager can invoke the method in the same process as other routines. In general, to be threadsafe, a method should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED methods can be THREADSAFE.
- If the method is defined as NOT THREADSAFE, the database manager will never invoke the method in the same process as another routine.

For FENCED methods, THREADSAFE is the default if the LANGUAGE is JAVA. For all other languages, NOT THREADSAFE is the default. If the method is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).

For NOT FENCED methods, THREADSAFE is the default. NOT THREADSAFE cannot be specified (SQLSTATE 42613).

RETURNS NULL ON NULL INPUT or **CALLED ON NULL INPUT**
This optional clause may be used to avoid a call to the external method if any of the non-subject arguments is null.

If RETURNS NULL ON NULL INPUT is specified, and if at execution time any one of the method's arguments is null, the method is not called and the result is the null value.

If CALLED ON NULL INPUT is specified, then regardless of the number of null arguments, the method is called. It can return a null value or a normal (non-null) value. However, responsibility for testing for null argument values lies with the method.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

There are two cases in which this specification is ignored:

- If the subject argument is null, in which case the method is not executed and the result is null
- If the method is defined to have no parameters, in which case this null argument condition cannot occur.

NO SQL, CONTAINS SQL, READS SQL DATA
Indicates whether the method issues any SQL statements and, if so, what type.

NO SQL
Indicates that the method cannot execute any SQL statements (SQLSTATE 38001).

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the method (SQLSTATE 38004 or 42985). Statements that are not supported in any method return a different error (SQLSTATE 38003 or 42985).

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the method (SQLSTATE 38002 or 42985). Statements that are not supported in any method return a different error (SQLSTATE 38003 or 42985).

EXTERNAL ACTION or **NO EXTERNAL ACTION**
This optional clause specifies whether or not the method takes some action that changes the state of an object not managed by the database manager. Optimizations that assume methods have no external impacts are prevented by specifying EXTERNAL ACTION.

NO SCRATCHPAD or **SCRATCHPAD** *length*
This optional clause may be used to specify whether a scratchpad is to be provided for an external method. It is strongly recommended that methods be re-entrant, so a scratchpad provides a means for the method to "save state" from one call to the next.

If SCRATCHPAD is specified, then at the first invocation of the user-defined method, memory is allocated for a scratchpad to be used by the external method. This scratchpad has the following characteristics:

- *length*, if specified, sets the size in bytes of the scratchpad and must be between 1 and 32 767 (SQLSTATE 42820). The default value is 100.
- It is initialized to all X'00''s.
- Its scope is the SQL statement. There is one scratchpad per reference to the external method in the SQL statement.

So, if method X in the following statement is defined with the SCRATCHPAD keyword, three scratchpads would be assigned.

```
SELECT A, X..(A) FROM TABLEB
   WHERE X..(A) > 103 OR X..(A) < 19
```

If ALLOW PARALLEL is specified or defaulted to, then the scope is different from the above. If the method is executed in multiple partitions, a scratchpad would be assigned in each partition where the method is processed, for each reference to the method in the SQL

statement. Similarly, if the query is executed with intra-partition parallelism enabled, more than three scratchpads may be assigned.

The scratchpad is persistent. Its content is preserved from one external method call to the next. Any changes made to the scratchpad by the external method on one call will be present on the next call. The database manager initializes scratchpads at the beginning of execution of each SQL statement. The database manager may reset scratchpads at the beginning of execution of each subquery. The system issues a final call before resetting a scratchpad if the FINAL CALL option is specified.

The scratchpad can be used as a central point for system resources (memory, for example) which the external method might acquire. The method could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

In such a case where system resource is acquired, the FINAL CALL keyword should also be specified; this causes a special call to be made at end-of-statement to allow the external method to free any system resources acquired.

If SCRATCHPAD is specified, then on each invocation of the user-defined method, an additional argument is passed to the external method which addresses the scratchpad.

If NO SCRATCHPAD is specified, then no scratchpad is allocated or passed to the external method.

NO FINAL CALL or **FINAL CALL**
This optional clause specifies whether a final call is to be made to an external method. The purpose of such a final call is to enable the external method to free any system resources it has acquired. It can be useful in conjunction with the SCRATCHPAD keyword in situations where the external method acquires system resources such as memory and anchors them in the scratchpad.

If FINAL CALL is specified, then at execution time, an additional argument is passed to the external method which specifies the type of call. The types of calls are:

- Normal call: SQL arguments are passed and a result is expected to be returned.
- First call: the first call to the external method for this specific reference to the method in this specific SQL statement. The first call is a normal call.

- Final call: a final call to the external method to enable the method to free up resources. The final call is not a normal call. This final call occurs at the following times:
  - End-of-statement: this case occurs when the cursor is closed for cursor-oriented statements, or when the statement is through executing otherwise.
  - End-of-transaction: This case occurs when the normal end-of-statement does not occur. For example, the logic of an application may for some reason bypass the close of the cursor.

  If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made at the subsequent close of the cursor or at the end of the application.

If NO FINAL CALL is specified, then no "call type" argument is passed to the external method, and no final call is made.

ALLOW PARALLEL or **DISALLOW PARALLEL**

This optional clause specifies whether, for a single reference to the method, the invocation of the method can be parallelized. In general, the invocations of most scalar methods should be parallelizable, but there may be methods (such as those depending on a single copy of a scratchpad) that cannot. If either ALLOW PARALLEL or DISALLOW PARALLEL are specified for a method, then DB2 will accept this specification.

The following questions should be considered in determining which keyword is appropriate for the method:.

- Are all the method invocations completely independent of each other? If YES, then specify ALLOW PARALLEL.
- Does each method invocation update the scratchpad, providing value(s) that are of interest to the next invocation (the incrementing of a counter, for example)? If YES, then specify DISALLOW PARALLEL or accept the default.
- Is there some external action performed by the method which should happen only on one partition? If YES, then specify DISALLOW PARALLEL or accept the default.
- Is the scratchpad used, but only so that some expensive initialization processing can be performed a minimal number of times? If YES, then specify ALLOW PARALLEL.

In any case, the body of every external method should be in a directory that is available on every partition of the database.

The syntax diagram indicates that the default value is ALLOW PARALLEL. However, the default is DISALLOW PARALLEL if one or more of the following options is specified in the statement:

- NOT DETERMINISTIC
- EXTERNAL ACTION
- SCRATCHPAD
- FINAL CALL

NO DBINFO or **DBINFO**
This optional clause specifies whether certain specific information known by DB2 will be passed to the method as an additional invocation-time argument (DBINFO), or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613). If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

If DBINFO is specified, a structure that contains the following information is passed to the method:

- Database name - the name of the currently connected database.
- Application ID - unique application ID which is established for each connection to the database.
- Application Authorization ID - the application runtime authorization ID, regardless of the nested methods in between this method and the application.
- Code page - identifies the database code page.
- Schema name - under the exact same conditions as for Table name, contains the name of the schema; otherwise blank.
- Table name - if and only if the method reference is either the right-hand side of a SET clause in an UPDATE statement, or an item in the VALUES list of an INSERT statement, contains the unqualified name of the table being updated or inserted; otherwise blank.
- Column name - under the exact same conditions as for Table name, contains the name of the column being updated or inserted; otherwise blank.
- Database version/release - identifies the version, release and modification level of the database server invoking the method.
- Platform - contains the server's platform type.
- Table method result column numbers - not applicable to methods.

INHERIT SPECIAL REGISTERS
This optional clause specifies that special registers in the method will inherit their initial values from the calling statement. For cursors, the initial values are inherited from the time that the cursor is opened.

No changes to the special registers are passed back to the caller of the method.

Some special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore never inherited from the caller.

NOT FEDERATED
This optional clause specifies whether nicknames cannot be used.

If NOT FEDERATED is specified, federated objects cannot be used in any SQL statement in the method. Using a nickname will result in an error (SQLSTATE 55047).

Notes:

- ***Compatibilities***
  - For compatibility with DB2 UDB for OS/390 and z/OS:
    - The following syntax is tolerated:
      - NOT VARIANT can be specified in place of DETERMINISTIC
      - VARIANT can be specified in place of NOT DETERMINISTIC
      - NULL CALL can be specified in place of CALLED ON NULL INPUT
      - NOT NULL CALL can be specified in place of RETURNS NULL ON NULL INPUT
  - For compatibility with previous versions of DB2:
    - PARAMETER STYLE DB2SQL can be specified in place of PARAMETER STYLE SQL
- Creating a structured type with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- A structured subtype defined with no attributes defines a subtype that inherits all its attributes from the supertype. If neither an UNDER clause nor any other attribute is specified, then the type is a root type of a type hierarchy without any attributes.
- The addition of a new subtype to a type hierarchy may cause packages to be invalidated. A package may be invalidated if it depends on a supertype of the new type. Such a dependency is the result of the use of a TYPE predicate or a TREAT specification.
- A structured type may have no more than 4082 attributes (SQLSTATE 54050).
- A method specification is not allowed to have the same signature as a function (comparing the first parameter-type of the function with the subject-type of the method).
- No original method may override another method, or be overridden by an original method (SQLSTATE 42745). Furthermore, a function and a method

cannot be in an overriding relationship. This means that if the function were considered to be a method with its first parameter as subject S, it must not override another method in any supertype of S, and it must not be overridden by another method in any subtype of S (SQLSTATE 42745).

- Creation of a structured type automatically generates a set of functions and methods for use with the type. All the functions and methods are generated in the same schema as the structured type. If the signature of the generated function or method conflicts with or overrides the signature of an existing function in this schema, the statement fails (SQLSTATE 42710). The generated functions or methods cannot be dropped without dropping the structured type (SQLSTATE 42917). The following functions and methods are generated:
  - Functions
    - Reference Comparisons

      Six comparison functions with names =, <>, <, <=, >, >= are generated for the reference type REF(*type-name*). Each of these functions takes two parameters of type REF(*type-name*) and returns true, false, or unknown. The comparison operators for REF(*type-name*) are defined to have the same behavior as the comparison operators for the underlying data type of REF(*type-name*). (All references in a type hierarchy have the same reference representation type. This enables REF(S) and REF(T) to be compared, provided that S and T have a common supertype. Because uniqueness of the OID column is enforced only within a table hierarchy, it is possible that a value of REF(T) in one table hierarchy may be "equal" to a value of REF(T) in another table hierarchy, even though they reference different rows.)

      The scope of the reference type is not considered in the comparison.

    - Cast functions

      Two cast functions are generated to cast between the generated reference type REF(*type-name*) and the underlying data type of this reference type.

      - The name of the function to cast from the underlying type to the reference type is the implicit or explicit *funcname1*.

        The format of this function is:

        ```
        CREATE FUNCTION funcname1 (rep-type)
          RETURNS REF(type-name) ...
        ```

      - The name of the function to cast from the reference type to the underlying type of the reference type is the implicit or explicit *funcname2*.

        The format of this function is:

        ```
        CREATE FUNCTION funcname2 ( REF(type-name) )
          RETURNS rep-type ...
        ```

For some rep-types, there are additional cast functions generated with *funcname1* to handle casting from constants.

- If *rep-type* is SMALLINT, the additional generated cast function has the format:

```
CREATE FUNCTION funcname1 (INTEGER)
  RETURNS REF(type-name)
```

- If *rep-type* is CHAR(n), the additional generated cast function has the format:

```
CREATE FUNCTION funcname1 ( VARCHAR(n))
  RETURNS REF(type-name)
```

- If *rep-type* is GRAPHIC(n), the additional generated cast function has the format:

```
CREATE FUNCTION funcname1 (VARGRAPHIC(n))
  RETURNS REF(type-name)
```

The schema name of the structured type must be included in the SQL path for successful use of these operators and cast functions in SQL statements.

- Constructor function

  The constructor function is generated to allow a new instance of the type to be constructed. This new instance will have null for all attributes of the type, including attributes that are inherited from a supertype.

  The format of the generated constructor function is:

```
CREATE FUNCTION type-name ( )
  RETURNS type-name
  ...
```

  If NOT INSTANTIABLE is specified, no constructor function is generated. If the structured type has attributes of type DATALINK, then the invocation of the constructor function fails (SQLSTATE 428ED).

– Methods

  - Observer methods

    An observer method is defined for each attribute of the structured type. For each attribute, the observer method returns the type of the attribute. If the subject is null, the observer method returns a null value of the attribute type.

    For example, the attributes of an instance of the structured type ADDRESS can be observed using `C1..STREET`, `C1..CITY`, `C1..COUNTRY`, and `C1..CODE`.

    The method signature of the generated observer method is as if the following statement had been executed:

```
CREATE TYPE  type-name
   ...
  METHOD attribute-name()
    RETURNS attribute-type
```

where *type-name* is the structured type name.

- Mutator methods

  A type-preserving mutator method is defined for each attribute of the structured type. Use mutator methods to change attributes within an instance of a structured type. For each attribute, the mutator method returns a copy of the subject modified by assigning the argument to the named attribute of the copy.

  For example, an instance of the structured type ADDRESS can be mutated using `C1..CODE('M3C1H7')`. If the subject is null, the mutator method raises an error (SQLSTATE 2202D).

  The method signature of the generated mutator method is as if the following statement had been executed:

```
CREATE TYPE  type-name
     ...
   METHOD attribute-name (attribute-type)
      RETURNS type-name
```

  If the attribute data type is SMALLINT, REAL, CHAR, or GRAPHIC, an additional mutator method is generated in order to support mutation using constants:

  - If *attribute-type* is SMALLINT, the additional mutator supports an argument of type INTEGER.
  - If *attribute-type* is REAL, the additional mutator supports an argument of type DOUBLE.
  - If *attribute-type* is CHAR, the additional mutator supports an argument of type VARCHAR.
  - If *attribute-type* is GRAPHIC, the additional mutator supports an argument of type VARGRAPHIC.

- If the structured type is used as a column type, the length of an instance of the type can be no more than 1 GB in length at runtime (SQLSTATE 54049).

- When creating a new subtype for an existing structured type (for use as a column type), any transform functions already written in support of existing related structured types should be re-examined and updated as necessary. Whether the new type is in the same hierarchy as a given type, or in the hierarchy of a nested type, it is likely that the existing transform function associated with this type will need to be modified to include some or all of the new attributes introduced by the new subtype. Generally speaking, because it is the set of transform functions associated with a

given type (or type hierarchy) that enables UDF and client application access to the structured type, the transform functions should be written to support *all* of the attributes in a given composite hierarchy (that is, including the transitive closure of all subtypes and their nested structured types).

When a new subtype of an existing type is created, all packages dependent on methods that are defined in supertypes of the type being created, and that are eligible for overriding, are invalidated.

- ***Table access restrictions***

  If a method is defined as READS SQL DATA, no statement in the method can access a table that is being modified by the statement which invoked the method (SQLSTATE 57053). For example, suppose the method BONUS() is defined as READS SQL DATA. If the statement UPDATE DEPTINFO SET SALARY = SALARY + EMP..BONUS() is invoked, no SQL statement in the BONUS method can read from the EMPLOYEE table.

- ***Privileges***
  - The definer of the user-defined type always receives the EXECUTE privilege WITH GRANT OPTION on all methods and functions automatically generated for the structured type. The EXECUTE privilege is not granted on any methods explicitly specified in the CREATE TYPE statement until a method body is defined using the CREATE METHOD statement. The definer of the user-defined type does have the right to drop the method specification using the ALTER TYPE statement. EXECUTE privilege on all functions automatically generated during the CREATE DISTINCT TYPE is granted to PUBLIC.
  - When an external method is used in an SQL statement, the method definer must have the EXECUTE privilege on any packages used by the method.
- Only routines defined as NO SQL can be used to define an index extension (SQLSTATE 428F8).

Examples:

Example 1: Create a type for department.

```
CREATE TYPE DEPT AS
   (DEPT NAME     VARCHAR(20),
      MAX_EMPS INT)
      REF USING INT
    MODE DB2SQL
```

Example 2: Create a type hierarchy consisting of a type for employees and a subtype for managers.

```
CREATE TYPE EMP AS
  (NAME      VARCHAR(32),
  SERIALNUM INT,
```

```
  DEPT      REF(DEPT),
  SALARY    DECIMAL(10,2))
  MODE DB2SQL

CREATE TYPE MGR UNDER EMP AS
  (BONUS     DECIMAL(10,2))
  MODE DB2SQL
```

Example 3: Create a type hierarchy for addresses. Addresses are intended to be used as types of columns. The inline length is not specified, so DB2 will calculate a default length. Encapsulate within the address type definition an external method that calculates how close this address is to a given input address. Create the method body using the CREATE METHOD statement.

```
CREATE TYPE address_t AS
  (STREET     VARCHAR(30),
  NUMBER     CHAR(15),
  CITY       VARCHAR(30),
  STATE      VARCHAR(10))
  NOT FINAL
  MODE DB2SQL
    METHOD SAMEZIP (addr address_t)
    RETURNS INTEGER
    LANGUAGE SQL
    DETERMINISTIC
    CONTAINS SQL
    NO EXTERNAL ACTION

    METHOD DISTANCE (address_t)
    RETURNS FLOAT
    LANGUAGE C
    DETERMINISTIC
    PARAMETER STYLE SQL
    NO SQL
    NO EXTERNAL ACTION

CREATE TYPE germany_addr_t UNDER address_t AS
  (FAMILY_NAME VARCHAR(30))
  NOT FINAL
  MODE DB2SQL

CREATE TYPE us_addr_t UNDER address_t AS
  (ZIP VARCHAR(10))
  NOT FINAL
  MODE DB2SQL
```

Example 4: Create a type that has nested structured type attributes.

```
CREATE TYPE PROJECT AS
  (PROJ_NAME  VARCHAR(20),
   PROJ_ID    INTEGER,
   PROJ_MGR   MGR,
```

```
PROJ_LEAD  EMP,
LOCATION   ADDR_T,
AVAIL_DATE DATE)
MODE DB2SQL
```

Related reference:

- "Basic predicate" in the *SQL Reference, Volume 1*
- "CREATE TABLE" on page 332
- "SET PATH" on page 726
- "Special registers" in the *SQL Reference, Volume 1*

Related samples:

- "dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)"

CREATE TYPE MAPPING

The CREATE TYPE MAPPING statement creates a mapping between these data types:

- A data type of a column of a data source table or view that is going to be defined to a federated database
- A corresponding data type that is already defined to the federated database.

The mapping can associate the federated database data type with a data type at either (1) a specified data source or (2) a range of data sources; for example, all data sources of a particular type and version.

A data type mapping must be created only if an existing one is not adequate.

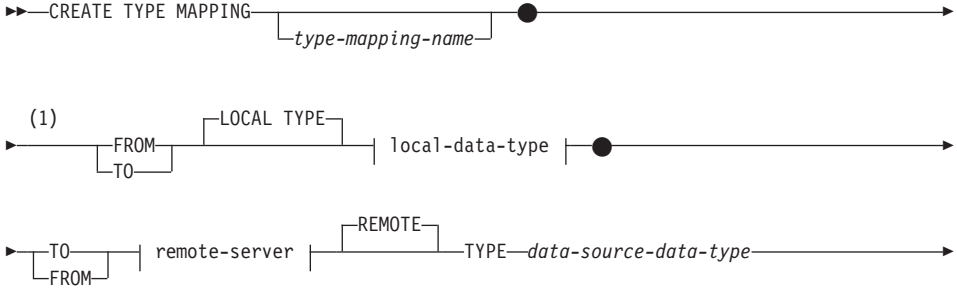
Invocation:

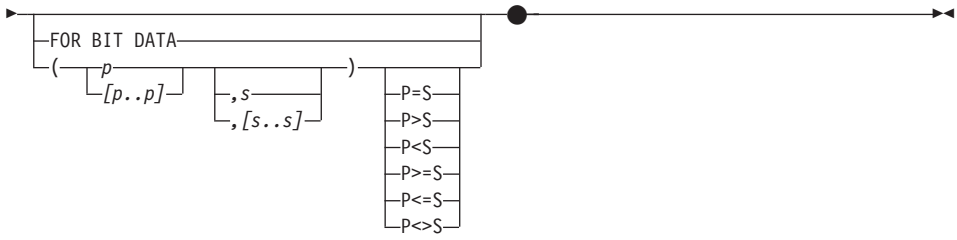
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

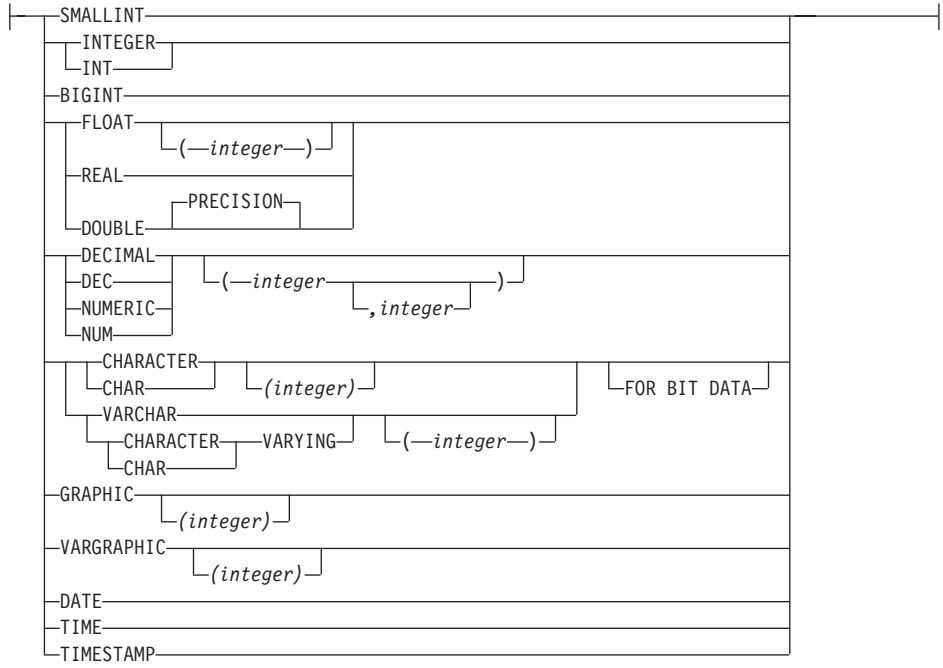
The privileges held by the authorization ID of the statement must have SYSADM or DBADM authority.

Syntax:



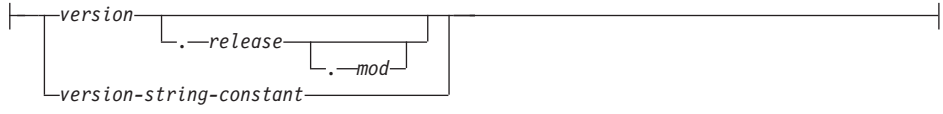


local-data-type:



remote-server:

server-version:



Notes:

1 Both a TO and a FROM keyword must be present in the CREATE TYPE MAPPING statement.

Description:

type-mapping-name
Names the data type mapping. The name must not identify a data type mapping that is already described in the catalog. A unique name is generated if *type-mapping-name* is not specified.

FROM or **TO**
Specifies whether a remote (data source) data type is to be mapped to a local DB2 data type (forward type mapping), or a local DB2 data type is to be mapped to a remote data type (reverse type mapping).

local-data-type
Identifies a data type that is defined to a federated database. If *local-data-type* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). If length or precision (and scale) are not specified for the *local-data-type*, then the values are determined from the *source-data-type*.

The *local-data-type* cannot be LONG VARCHAR, LONG VARGRAPHIC, DATALINK, a large object (LOB) type, or a user-defined type (SQLSTATE 42806).

SERVER *server-name*
Names the data source to which *data-source-data-type* is defined.

SERVER TYPE *server-type*
Identifies the type of data source to which *data-source-data-type* is defined.

VERSION
Identifies the version of the data source to which *data-source-data-type* is defined.

version
Specifies the version number. *version* must be an integer.

release
Specifies the number of the release of the version denoted by *version*; *release* must be an integer.

mod
Specifies the number of the modification of the release denoted by *release*; *mod* must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release* and, if applicable, *mod* (for example, '8.0.3').

WRAPPER *wrapper-name*
Specifies the name of the wrapper that the federated server uses to interact with data sources of the type and version denoted by *server-type* and *server-version*.

TYPE *data-source-data-type*
Specifies the data source data type that is being mapped to *local-data-type*. If *data-source-data-type* is qualified by a schema name at the data source, it is allowable, but not required, to specify this qualifier.

The *data-source-data-type* must be a built-in data type. User-defined types are not allowed. If the type has a short and long form (for example, CHAR and CHARACTER), the short form should be specified.

p For a decimal data type, *p* specifies the maximum number of digits that a value can have. For all other data types for character data, *p* specifies the maximum number of characters that a value can have. The *p* must be valid with respect to the data type (SQLSTATE 42611). If *p* is not specified and the data type requires it, the system will determine the best match.

[p..p]
For a decimal data type, *[p..p]* specifies the minimum and maximum number of digits that a value can have. For all other data types for character data, *[p..p]* specifies the minimum and maximum number of characters that a value can have. In all cases, the maximum must equal or exceed the minimum; and both numbers must be valid with respect to the data type (SQLSTATE 42611).

s For a decimal data type, *s* specifies the allowable maximum number of digits to the right of the decimal point. This number must be valid with respect to the data type (SQLSTATE 42611). If a number is not specified and the data type requires one, the system will determine the best match.

[s..s]
For a decimal data type, *[s..s]* specifies the minimum and maximum number of digits allowed to the right of the decimal point. The maximum must equal or exceed the minimum, and both numbers must be valid with respect to the data type (SQLSTATE 42611).

P [operand] **S**
For a decimal data type, P [operand] S specifies a comparison between the maximum allowable precision and the maximum number of digits allowed to the right of the decimal point. For example, the operand = indicates that the maximum allowable precision and the maximum

number digits allowed in the decimal fraction are the same. Specify P [operand] S only if the level of checking that it enforces is required.

FOR BIT DATA
Indicates whether *data-source-data-type* is for bit data. These keywords are required if the data source type column contains binary values. The database manager will determine this attribute if it is not specified on a character data type.

Notes:

A CREATE TYPE MAPPING statement within a given unit of work (UOW) cannot be processed under either of the following conditions:

- The statement references a single data source, and the UOW already includes a SELECT statement that references a nickname for a table or view within this data source.
- The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes a SELECT statement that references a nickname for a table or view within one of these data sources.

Examples:

Example 1: Create a mapping between SYSIBM.DATE and the Oracle data type DATE at all Oracle data sources.

```
CREATE TYPE MAPPING MY_ORACLE_DATE
  FROM SYSIBM.DATE
  TO SERVER TYPE ORACLE
  TYPE DATE
```

Example 2: Create a mapping between SYSIBM.DECIMAL(10,2) and the Oracle data type NUMBER([10..38],2) at data source ORACLE1.

```
CREATE TYPE MAPPING MY_ORACLE_DEC
  FROM SYSIBM.DECIMAL(10,2)
  TO SERVER ORACLE1
  TYPE NUMBER([10..38],2)
```

CREATE USER MAPPING

The CREATE USER MAPPING statement defines a mapping between an authorization ID that uses a federated database and the authorization ID and password to use at a specified data source.

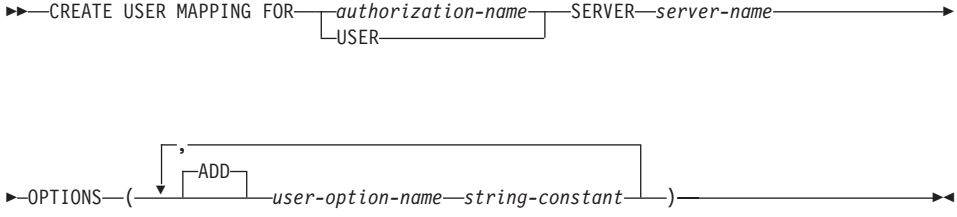
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

If the authorization ID of the statement is different than the authorization name that is being mapped to the data source, then the authorization ID must include SYSADM or DBADM authority. Otherwise, if the authorization ID and the authorization name match, then no privileges or authorities are required.

Syntax:



Description:

authorization-name
Specifies the authorization name under which a user or application connects to a federated database. This name is to be mapped to an identifier under which the data source denoted by *server-name* can be accessed.

USER
The value in the special register USER. When USER is specified, then the authorization ID of the CREATE USER MAPPING statement will be mapped to the data source authorization ID that is specified in the REMOTE_AUTHID user option.

SERVER *server-name*
Identifies the data source that is accessible under the mapping authorization ID.

OPTIONS
Indicates what user options are to be enabled.

ADD
Enables one or more user options.

user-option-name
Names a user option that will be used to complete the user mapping that is being created.

string-constant
Specifies the setting for *user-option-name* as a character string constant.

Notes:

- A user mapping cannot be created in a given unit of work (UOW) if the UOW already includes a SELECT statement that references a nickname for a table or view at the data source that is to be included in the mapping.

Examples:

Example 1: To access a data source called S1, you need to map your authorization name and password for your local database to your user ID and password for S1. Your authorization name is RSPALTEN, and the user ID and password that you use for S1 are SYSTEM and MANAGER, respectively.

```
CREATE USER MAPPING FOR RSPALTEN
    SERVER S1
    OPTIONS
    ( REMOTE_AUTHID 'SYSTEM',
      REMOTE_PASSWORD 'MANAGER' )
```

Example 2: Marc already has access to a DB2 data source. He now needs access to an Oracle data source, so that he can create joins between certain DB2 and Oracle tables. He acquires a user name and password for the Oracle data source; the user name is the same as his authorization ID for the federated database, but his Oracle and federated database passwords are different. To be able to access Oracle from the federated database, he must map the two passwords together.

```
CREATE USER MAPPING FOR MARCR
    SERVER ORACLE1
    OPTIONS
    ( REMOTE_PASSWORD 'NZXCZY' )
```

Related reference:

- "User options for federated systems" in the *Federated Systems Guide*

CREATE VIEW

The CREATE VIEW statement creates a view on one or more tables, views or nicknames.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority or
- For each table, view or nickname identified in any fullselect:
  - CONTROL privilege on that table or view, or
  - SELECT privilege on that table or view

  and at least one of the following:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the view does not exist
  - CREATEIN privilege on the schema, if the schema name of the view refers to an existing schema.

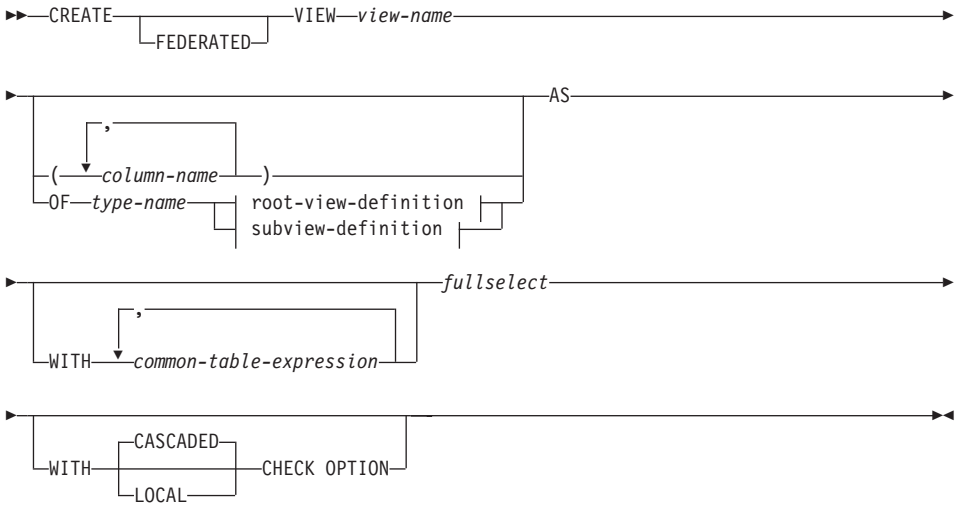
  If creating a subview, the authorization ID of the statement must:
  - be the same as the definer of the root table of the table hierarchy.
  - have SELECT WITH GRANT on the underlying table of the subview or the superview must not have SELECT privilege granted to any user other than the view definer.

Group privileges are not considered for any table or view specified in the CREATE VIEW statement.

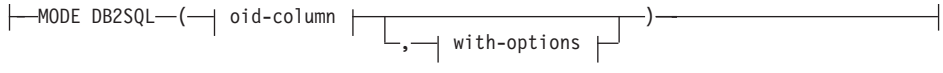
Privileges are not considered when defining a view on federated database nickname. Authorization requirements of the data source for the table or view referenced by the nickname are applied when the query is processed. The authorization ID of the statement may be mapped to a different remote authorization ID.

If a view definer can only create the view because the definer has SYSADM authority, then the definer is granted explicit DBADM authority for the purpose of creating the view.

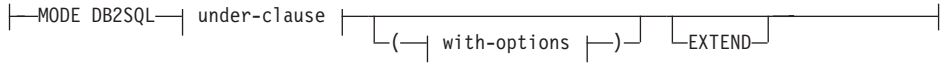
Syntax:



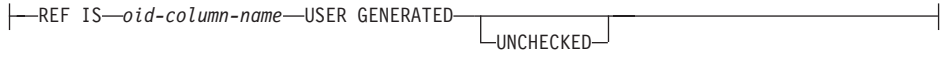
root-view-definition:



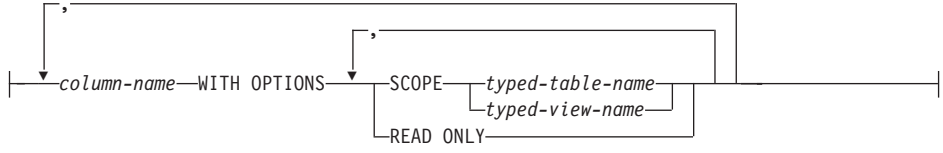
subview-definition:



oid-column:



with-options:



under-clause:

Description:

FEDERATED
Indicates that the view being created references a nickname or an OLEDB table function. If an OLEDB table function or a nickname is directly, or indirectly, referenced in the fullselect and the FEDERATED keyword is not specified, a warning will be issued (SQLSTATE 01639) when the CREATE VIEW statement is submitted. However, the view will still be created.

Conversely, if an OLEDB table function or a nickname is not directly, or indirectly, referenced in the fullselect and the FEDERATED keyword is specified, an error will be issued (SQLSTATE 429BA) when the CREATE VIEW statement is submitted. The view will not be created.

view-name
Names the view. The name, including the implicit or explicit qualifier, must not identify a table, view, nickname or alias described in the catalog. The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

The name can be the same as the name of an inoperative view (see "Inoperative views" on page 473). In this case the new view specified in the CREATE VIEW statement will replace the inoperative view. The user will get a warning (SQLSTATE 01595) when an inoperative view is replaced. No warning is returned if the application was bound with the bind option SQLWARN set to NO.

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 (SQLSTATE 42908). An unnamed column is a column derived from a constant, function, expression, or set operation that is not named using the AS clause of the select list.

OF *type-name*
Specifies that the columns of the view are based on the attributes of the structured type identified by *type-name*. If *type-name* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). The type name

must be the name of an existing user-defined type (SQLSTATE 42704) and it must be a structured type that is instantiable (SQLSTATE 428DP).

MODE DB2SQL
This clause is used to specify the mode of the typed view. This is the only valid mode currently supported.

UNDER *superview-name*
Indicates that the view is a subview of *superview-name*. The superview must be an existing view (SQLSTATE 42704) and the view must be defined using a structured type that is the immediate supertype of *type-name* (SQLSTATE 428DB). The schema name of *view-name* and *superview-name* must be the same (SQLSTATE 428DQ). The view identified by *superview-name* must not have any existing subview already defined using *type-name* (SQLSTATE 42742).

The columns of the view include the object identifier column of the superview with its type modified to be REF(*type-name*), followed by columns based on the attributes of *type-name* (remember that the type includes the attributes of its supertype).

INHERIT SELECT PRIVILEGES
Any user or group holding a SELECT privilege on the superview will be granted an equivalent privilege on the newly created subview. The subview definer is considered to be the grantor of this privilege.

OID-column
Defines the object identifier column for the typed view.

REF IS *OID-column-name* **USER GENERATED**
Specifies that an object identifier (OID) column is defined in the view as the first column. An OID is required for the root view of a view hierarchy (SQLSTATE 428DX). The view must be a typed view (the OF clause must be present) that is not a subview (SQLSTATE 42613). The name for the column is defined as *OID-column-name* and cannot be the same as the name of any attribute of the structured type *type-name* (SQLSTATE 42711). The first column specified in *fullselect* must be of type REF(*type-name*) (you may need to cast it so that it has the appropriate type). If UNCHECKED is not specified, it must be based on a not nullable column on which uniqueness is enforced through an index (primary key, unique constraint, unique index, or OID-column). This column will be referred to as the *object identifier column* or *OID column*. The keywords USER GENERATED indicate that the initial value for the OID column must be provided by the user when inserting a row. Once a row is inserted, the OID column cannot be updated (SQLSTATE 42808).

UNCHECKED
Defines the object identifier column of the typed view definition to

assume uniqueness even though the system can not prove this uniqueness. This is intended for use with tables or views that are being defined into a typed view hierarchy where the user knows that the data conforms to this uniqueness rule but it does not comply with the rules that allow the system to prove uniqueness. UNCHECKED option is mandatory for view hierarchies that range over multiple hierarchies or legacy tables or views By specifying UNCHECKED, the user takes responsibility for ensuring that each row of the view has a unique OID. If the user fails to ensure this property, and a view contains duplicate OID values, then a path-expression or DEREF operator involving one of the non-unique OID values may result in an error (SQLSTATE 21000).

with-options
Defines additional options that apply to columns of a typed view.

column-name **WITH OPTIONS**
Specifies the name of the column for which additional options are specified. The *column-name* must correspond to the name of an attribute defined in (not inherited by) the *type-name* of the view. The column must be a reference type (SQLSTATE 42842). It cannot correspond to a column that also exists in the superview (SQLSTATE 428DJ). A column name can only appear in one WITH OPTIONS SCOPE clause in the statement (SQLSTATE 42613).

SCOPE
Identifies the scope of the reference type column. A scope must be specified for any column that is intended to be used as the left operand of a dereference operator or as the argument of the DEREF function.

Specifying the scope for a reference type column may be deferred to a subsequent ALTER VIEW statement (if the scope is not inherited) to allow the target table or view to be defined, usually in the case of mutually referencing views and tables. If no scope is specified for a reference type column of the view and the underlying table or view column was scoped, then the underlying column's scope is inherited by the reference type column. The column remains unscoped if the underlying table or view column did not have a scope. See 471 for more information about scope and reference type columns.

typed-table-name
The name of a typed table. The table must already exist or be the same as the name of the table being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

typed-view-name
The name of a typed view. The view must already exist or be the same as the name of the view being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

READ ONLY
Identifies the column as a read-only column. This option is used to force a column to be read-only so that subview definitions can specify an expression for the same column that is implicitly read-only.

AS
Identifies the view definition.

WITH *common-table-expression*
Defines a common table expression for use with the fullselect that follows. A common table expression cannot be specified when defining a typed view.

fullselect
Defines the view. At any time, the view consists of the rows that would result if the SELECT statement were executed. The fullselect must not reference host variables, parameter markers, or declared temporary tables. However, a parameterized view can be created as an SQL table function.

For Typed Views and Subviews: The *fullselect* must conform to the following rules otherwise an error is returned (SQLSTATE 428EA unless otherwise specified).

- The fullselect must not include references to the DBPARTITIONNUM or HASHEDVALUE functions, non-deterministic functions, or functions defined to have external action.
- The body of the view must consist of a single subselect, or a UNION ALL of two or more subselects. Let each of the subselects participating directly in the view body be called a *branch* of the view. A view may have one or more branches.
- The FROM-clause of each branch must consist of a single table or view (not necessarily typed), called the *underlying* table or view of that branch.
- The underlying table or view of each branch must be in a separate hierarchy (i.e., a view may not have multiple branches with their underlying tables or views in the same hierarchy).
- None of the branches of a typed view definition may specify GROUP BY or HAVING.
- If the view body contains UNION ALL, then the root view in the hierarchy must specify the UNCHECKED option for its OID column.

For a hierarchy of views and subviews: Let BR1 and BR2 be any branches that appear in the definitions of views in the hierarchy. Let T1 be the underlying table or view of BR1, and let T2 be the underlying table or view of BR2. Then:

- If T1 and T2 are not in the same hierarchy, then the root view in the view hierarchy must specify the UNCHECKED option for its OID column.
- If T1 and T2 are in the same hierarchy, then BR1 and BR2 must contain predicates or ONLY-clauses that are sufficient to guarantee that their row-sets are disjoint.

For typed subviews defined using EXTEND AS: For every branch in the body of the subview:

- The underlying table of each branch must be a (not necessarily proper) subtable of some underlying table of the immediate superview.
- The expressions in the SELECT list must be assignable to the non-inherited columns of the subview (SQLSTATE 42854).

For typed subviews defined using AS without EXTEND:

- For every branch in the body of the subview, the expressions in the SELECT-list must be assignable to the declared types of the inherited and non-inherited columns of the subview (SQLSTATE 42854).
- The OID-expression of each branch over a given hierarchy in the subview must be equivalent (except for casting) to the OID-expression in the branch over the same hierarchy in the root view.
- The expression for a column not defined (implicitly or explicitly) as READ ONLY in a superview must be equivalent in all branches over the same underlying hierarchy in its subviews.

WITH CHECK OPTION

Specifies the constraint 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 does not satisfy the search conditions of the view.

WITH CHECK OPTION must not be specified if any of the following conditions is true:

- The view is read-only (SQLSTATE 42813). If WITH CHECK OPTION is specified for an updatable view that does not allow inserts, the constraint applies to updates only.
- The view references the NODENUMBER or PARTITION function, a non-deterministic function, or a function with external action (SQLSTATE 42997).
- A nickname is the update target of the view.

- A view that has an INSTEAD OF trigger defined on it is the update target of the view (SQLSTATE 428FQ).

If WITH CHECK OPTION is omitted, the definition of the view is not used in the checking of any insert or update operations that use the view. Some checking might still occur during insert or update operations if the view is directly or indirectly dependent on another view that includes WITH CHECK OPTION. 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.

CASCADED
The WITH CASCADED CHECK OPTION constraint on a view *V* means that *V* inherits the search conditions as constraints from any updatable view on which *V* is dependent. Furthermore, every updatable view that is dependent on *V* is also subject to these constraints. Thus, the search conditions of *V* and each view on which *V* is dependent are ANDed together to form a constraint that is applied for an insert or update of *V* or of any view dependent on *V*.

LOCAL
The WITH LOCAL CHECK OPTION constraint 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*.

The difference between CASCADED and LOCAL is shown in the following example. Consider the following updatable views (substituting for Y from column headings of the table that follows):

```
V1 defined on table T
V2 defined on V1 WITH Y CHECK OPTION
V3 defined on V2
V4 defined on V3 WITH Y CHECK OPTION
V5 defined on V4
```

The following table shows the search conditions against which inserted or updated rows are checked:

	Y is LOCAL	**Y is CASCADED**
V1 checked against:	no view	no view
V2 checked against:	V2	V2, V1
V3 checked against:	V2	V2, V1
V4 checked against:	V2, V4	V4, V3, V2, V1
V5 checked against:	V2, V4	V4, V3, V2, V1

Consider the following updatable view which shows the impact of the WITH CHECK OPTION using the default CASCADED option:

```
CREATE VIEW V1 AS SELECT COL1 FROM T1 WHERE COL1 > 10

CREATE VIEW V2 AS SELECT COL1 FROM V1 WITH CHECK OPTION

CREATE VIEW V3 AS SELECT COL1 FROM V2 WHERE COL1 < 100
```

The following INSERT statement using *V1* will succeed because *V1* does not have a WITH CHECK OPTION and *V1* is not dependent on any other view that has a WITH CHECK OPTION.

```
INSERT INTO V1 VALUES(5)
```

The following INSERT statement using *V2* will result in an error because *V2* has a WITH CHECK OPTION and the insert would produce a row that did not conform to the definition of *V2*.

```
INSERT INTO V2 VALUES(5)
```

The following INSERT statement using *V3* will result in an error even though it does not have WITH CHECK OPTION because *V3* is dependent on *V2* which does have a WITH CHECK OPTION (SQLSTATE 44000).

```
INSERT INTO V3 VALUES(5)
```

The following INSERT statement using *V3* will succeed even though it does not conform to the definition of *V3* (*V3* does not have a WITH CHECK OPTION); it does conform to the definition of *V2* which does have a WITH CHECK OPTION.

```
INSERT INTO V3 VALUES(200)
```

Notes:

- Creating a view with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- View columns inherit the NOT NULL WITH DEFAULT attribute from the base table or view except when columns are derived from an expression. When a row is inserted or updated into an updatable view, it is checked against the constraints (primary key, referential integrity, and check) if any are defined on the base table.
- A new view cannot be created if it uses an inoperative view in its definition. (SQLSTATE 51024).
- This statement does not support declared temporary tables (SQLSTATE 42995).
- ***Deletable views:*** A view is *deletable* if an INSTEAD OF trigger for the delete operation has been defined for the view, or if all of the following are true:

  - each FROM clause of the outer fullselect identifies only one base table (with no OUTER clause), deletable view (with no OUTER clause), deletable nested table expression, or deletable common table expression (cannot identify a nickname)
  - the outer fullselect does not include a VALUES clause
  - the outer fullselect does not include a GROUP BY clause or HAVING clause
  - the outer fullselect does not include column functions in the select list
  - the outer fullselect does not include SET operations (UNION, EXCEPT or INTERSECT) with the exception of UNION ALL
  - the base tables in the operands of a UNION ALL must not be the same table and each operand must be deletable
  - the select list of the outer fullselect does not include DISTINCT
- ***Updatable views:*** A column of a view is *updatable* if an INSTEAD OF trigger for the update operation has been defined for the view, or if all of the following are true:
  - the view is deletable (independent of an INSTEAD OF trigger for delete), the column resolves to a column of a base table (not using a dereference operation), and the READ ONLY option is not specified
  - all the corresponding columns of the operands of a UNION ALL have exactly matching data types (including length or precision and scale) and matching default values if the fullselect of the view includes a UNION ALL

  A view is updatable if *any* column of the view is updatable.
- ***Insertable views:***
  - A view is insertable if an INSTEAD OF trigger for the insert operation has been defined for the view, or at least one column of the view is updatable (independent of an INSTEAD OF trigger for update), and the fullselect of the view does not include UNION ALL.
  - A given row can be inserted into a view (including a UNION ALL) if, and only if, it fulfills the check constraints of exactly one of the underlying base tables.
  - To insert into a view that includes non-updatable columns, those columns must be omitted from the column list.
- ***Read-only views:*** A view is *read-only* if it is *not* deletable, updatable, or insertable.

  The READONLY column in the SYSCAT.VIEWS catalog view indicates if a view is read-only without considering INSTEAD OF triggers.
- Common table expressions and nested table expressions follow the same set of rules for determining whether they are deletable, updatable, insertable, or read-only.

- ***Inoperative views:*** An *inoperative view* is a view that is no longer available for SQL statements. A view becomes inoperative if:
  - A privilege, upon which the view definition is dependent, is revoked.
  - An object such as a table, nickname, alias or function, upon which the view definition is dependent, is dropped.
  - A view, upon which the view definition is dependent, becomes inoperative.
  - A view that is the superview of the view definition (the subview) becomes inoperative.

  In practical terms, an inoperative view is one in which the view definition has been unintentionally dropped. For example, when an alias is dropped, any view defined using that alias is made inoperative. All dependent views also become inoperative and packages dependent on the view are no longer valid.

  Until the inoperative view is explicitly recreated or dropped, a statement using that inoperative view cannot be compiled (SQLSTATE 51024) with the exception of the CREATE ALIAS, CREATE VIEW, DROP VIEW, and COMMENT ON TABLE statements. Until the inoperative view has been explicitly dropped, its qualified name cannot be used to create another table or alias (SQLSTATE 42710).

  An inoperative view may be recreated by issuing a CREATE VIEW statement using the definition text of the inoperative view. This view definition text is stored in the TEXT column of the SYSCAT.VIEWS catalog. When recreating an inoperative view, it is necessary to explicitly grant any privileges required on that view by others, due to the fact that all authorization records on a view are deleted if the view is marked inoperative. Note that there is no need to explicitly drop the inoperative view in order to recreate it. Issuing a CREATE VIEW statement with the same *view-name* as an inoperative view will cause that inoperative view to be replaced, and the CREATE VIEW statement will return a warning (SQLSTATE 01595).

  Inoperative views are indicated by an X in the VALID column of the SYSCAT.VIEWS catalog view and an X in the STATUS column of the SYSCAT.TABLES catalog view.

- ***Privileges***

  The definer of a view always receives the SELECT privilege on the view as well as the right to drop the view. The definer of a view will get CONTROL privilege on the view only if the definer has CONTROL privilege on every base table, view or nickname identified in the fullselect, or if the definer has SYSADM or DBADM authority.

The definer of the view is granted INSERT, UPDATE, column level UPDATE or DELETE privileges on the view if the view is not read-only and the definer has the corresponding privileges on the underlying objects.

The definer of a view only acquires privileges if the privileges from which they are derived exist at the time the view is created. The definer must have these privileges either directly or because PUBLIC has the privilege. Privileges are not considered when defining a view on federated server nickname. However, when using a view on a nickname, the user's authorization ID must have valid select privileges on the table or view that the nickname references at the data source. Otherwise, an error is returned. Privileges held by groups of which the view definer is a member, are not considered.

When a subview is created, the SELECT privileges held on the immediate superview are automatically granted on the subview.

- ***Scope and REF columns***

  When selecting a reference type column in the fullselect of a view definition, consider the target type and scope that is required.

  – If the required target type and scope is the same as the underlying table or view, the column can simply be selected.

  – If the scope needs to be changed, use the WITH OPTIONS SCOPE clause to define the required scope table or view.

  – If the target type of the reference needs to be changed, the column must be cast first to the representation type of the reference and then to the new reference type. The scope in this case can be specified in the cast to the reference type or using the WITH OPTIONS SCOPE clause. For example, assume you select column Y defined as REF(TYP1) SCOPE TAB1. You want this to be defined as REF(VTYP1) SCOPE VIEW1. The select list item would be as follows:

    ```
    CAST(CAST(Y AS VARCHAR(16) FOR BIT DATA) AS REF(VTYP1) SCOPE VIEW1)
    ```

- ***Identity columns*** A column of a view is considered an identity column, if the element of the corresponding column in the fullselect of the view definition is the name of an identity column of a table, or the name of a column of a view which directly or indirectly maps to the name of an identity column of a base table.

  In all other cases, the columns of a view will not get the identity property. For example:

  – the select-list of the view definition includes multiple instances of the name of an identity column (that is, selecting the same column more than once)

  – the view definition involves a join

  – a column in the view definition includes an expression that refers to an identity column

  – the view definition includes a UNION

When inserting into a view for which the select list of the view definition directly or indirectly includes the name of an identity column of a base table, the same rules apply as if the INSERT statement directly referenced the identity column of the base table.

- ***Federated views*** A federated view is a view that includes a reference to a nickname somewhere in the fullselect. The presence of such a nickname changes the authorization model used for the view both at create time and when the view is subsequently referenced in a query. If a view is created that references a nickname and the FEDERATED keyword is not included, a warning is issued to indicate that the authorization requirements for this view are different because of the reference to a nickname.

  A nickname has no associated DML privileges and therefore when the view is created, no privilege checking is done to determine whether the view definer has access to the nickname or to the underlying data source table or view. Privilege checking of references to tables or views at the federated database are handled as usual, requiring the view definer to have at least SELECT privilege on such objects.

  When a federated view is subsequently referenced in a query, the nicknames result in queries against the data source and authorization ID that issued the query (or the remote authorization ID to which it maps) must have the necessary privileges to access the data source table or view. The authorization ID that issues the query referencing the federated view is not required to have any additional privileges on tables or views (non-federated) that exist at the federated server.

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 SELECTPROJNO, 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 SELECTPROJNO, 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
```

Specifying the column name list could be avoided by naming the expression SALARY+BONUS+COMM as TOTAL_PAY in the fullselect.

```
CREATE VIEW PRJ_LEADER
  AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP,
              LASTNAME, SALARY+BONUS+COMM AS TOTAL_PAY
    FROM PROJECT, EMPLOYEE
    WHERE RESPEMP = EMPNO AND PRSTAFF > 1
```

Example 6: Given the set of tables and views shown in the following figure:

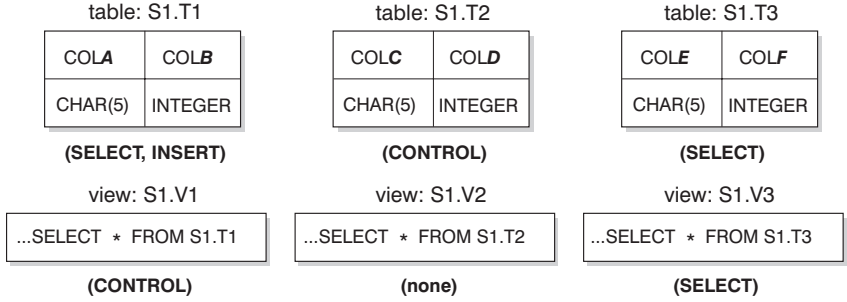


Figure 1. Tables and Views for Example 6

User ZORPIE (who does not have either DBADM or SYSADM authority) has been granted the privileges shown in brackets below each object:

1. ZORPIE will get CONTROL privilege on the view that she creates with:

   ```
   CREATE VIEW VA AS SELECT * FROM S1.V1
   ```

   because she has CONTROL on S1.V1. (CONTROL on S1.V1 must have been granted to ZORPIE by someone with DBADM or SYSADM authority.) It does not matter which, if any, privileges she has on the underlying base table.
2. ZORPIE will not be allowed to create the view:

   ```
   CREATE VIEW VB AS SELECT * FROM S1.V2
   ```

   because she has neither CONTROL nor SELECT on S1.V2. It does not matter that she has CONTROL on the underlying base table (S1.T2).
3. ZORPIE will get CONTROL privilege on the view that she creates with:

   ```
   CREATE VIEW VC (COLA, COLB, COLC, COLD)
     AS SELECT * FROM S1.V1, S1.T2
     WHERE COLA = COLC
   ```

   because the fullselect of ZORPIE.VC references view S1.V1 and table S1.T2 and she has CONTROL on both of these. Note that the view VC is read-only, so ZORPIE does not get INSERT, UPDATE or DELETE privileges.
4. ZORPIE will get SELECT privilege on the view that she creates with:

   ```
   CREATE VIEW VD (COLA,COLB, COLE, COLF)
     AS SELECT * FROM S1.V1, S1.V3
     WHERE COLA = COLE
   ```

   because the fullselect of ZORPIE.VD references the two views S1.V1 and S1.V3, one on which she has only SELECT privilege, and one on which she has CONTROL privilege. She is given the lesser of the two privileges, SELECT, on ZORPIE.VD.

5. ZORPIE will get INSERT, UPDATE and DELETE privilege WITH GRANT OPTION and SELECT privilege on the view VE in the following view definition.

```
CREATE VIEW VE
   AS SELECT * FROM S1.V1
  WHERE COLA > ANY
         (SELECT COLE FROM S1.V3)
```

   ZORPIE's privileges on VE are determined primarily by her privileges on S1.V1. Since S1.V3 is only referenced in a subquery, she only needs SELECT privilege on S1.V3 to create the view VE. The definer of a view only gets CONTROL on the view if they have CONTROL on all objects referenced in the view definition. ZORPIE does not have CONTROL on S1.V3, consequently she does not get CONTROL on VE.

Related concepts:

- "Queries" in the *SQL Reference, Volume 1*

Related reference:

- "CREATE FUNCTION (SQL Scalar, Table or Row)" on page 254

CREATE WRAPPER

The CREATE WRAPPER statement registers a wrapper to a federated database. A wrapper is a mechanism by which a federated server can interact with a certain category of data sources.

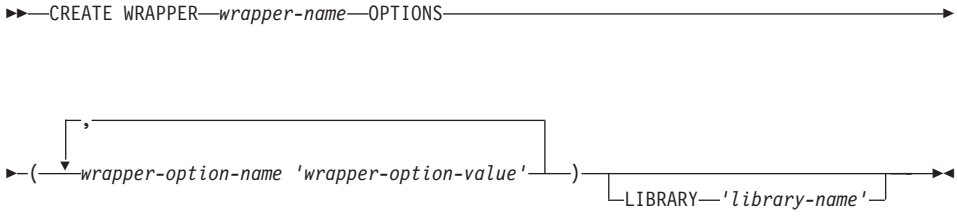
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement must have SYSADM or DBADM authority.

Syntax:



Description:

wrapper-name
Names the wrapper. It can be:

- A predefined name. If a predefined name is specified, the federated server automatically assigns a default to *'library-name'*.

  The predefined names are:

CTLIB	For Sybase data sources supported by Sybase Open Client Client-Library
DBLIB	For Sybase data sources supported by Sybase Open Client DB-Library
DJXMSSQL3	For all MS SQL Server Data sources that are supported by Microsoft ODBC SQL Server
DRDA	For all DB2 family data sources
INFORMIX	For all Informix data sources that are supported by Informix client SDK Version 2.x

MSSQLODBC3
: For all MS SQL Server data sources that are supported by DataDirect Technologies Connect ODBC for MS SQL Server

NET8
: For all Oracle data sources that are supported by Oracle's Net8 client software

OLEDB
: For all OLE DB providers supported by Microsoft OLE DB

SQLNET
: For all Oracle data sources that are supported by Oracle's SQL*Net client software

- A user-supplied name. If such a name is provided, it is necessary to also specify *'library-name'*.

OPTIONS (*wrapper-option-name 'wrapper-option-value', ...***)**
Currently, the only supported wrapper option name is DB2_FENCED; valid values for this option are 'Y' or 'N'. If wrapper options are not specified, then by default, the DB2_FENCED option is set to 'N'. It is recommended that the DB2_FENCED option be explicitly specified.

LIBRARY *'library-name'*
Names the file that contains the wrapper module.

The library name can be specified as an absolute path name or simply the base name (without the path). If only the base name is specified, the library should reside in the lib (UNIX) or the bin (Windows) subdirectory of the DB2 install path.

The LIBRARY option is only necessary when a user-supplied *wrapper-name* is used. This option should not be used when a predefined *wrapper-name* is given. The default library file names for the predefined *wrapper-names* are:

Table 10. Default file names for the LIBRARY option

	AIX	HP-UX	Linux	SOLARIS	WINNT
DJXMSSQL3	–	–	–	–	db2mssql3.dll
DRDA	libdb2drda.a	libdb2drda.sl	libdb2drda.so	libdb2drda.so	db2drda.dll
INFORMIX	libdb2informix.a	libdb2informix.sl	libdb2informix.so	libdb2informix.so	db2informix.dll
MSSQLODBC3	libmssql3.a	libmssql3.sl	libmssql3.so	libmssql3.so	–
NET8	libdb2net8.a	libdb2net8.sl	libdb2net8.so	libdb2net8.so	db2net8.dll
OLEDB	–	–	–	–	db2oledb.dll
SQLNET	libdb2sqlnet.a	–	–	–	db2sqlnet.dll

Examples:

Example 1: Register a wrapper that the federated server can use to interact with an Oracle data source that is supported by Oracle's SQL*Net client software. Use the predefined name.

```
CREATE WRAPPER SQLNET OPTIONS (DB2_FENCED 'N')
```

Example 2: Register a wrapper that the federated server on an AIX system can use to interact with DB2 for VM and VSE data sources. Specify a name to indicate that these data sources are used for testing.

```
CREATE WRAPPER TEST
  LIBRARY 'libsqlds.a' OPTIONS (DB2_FENCED 'N')
```

The extension in the library name (a) indicates that wrapper TEST is for data sources that reside in an AIX system.

Related reference:

- "ALTER WRAPPER" on page 97

DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

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 not an executable statement and cannot be dynamically prepared.

Authorization:

The term "SELECT statement of the cursor" is used in order to specify the authorization rules. The SELECT statement of the cursor is one of the following:

- The prepared select-statement identified by the *statement-name*
- The specified *select-statement*.

For each table or view identified (directly or using an alias) in the SELECT statement of the cursor, the privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority.
- For each table or view identified in the *select-statement*:
  - SELECT privilege on the table or view, or
  - CONTROL privilege of the table or view.

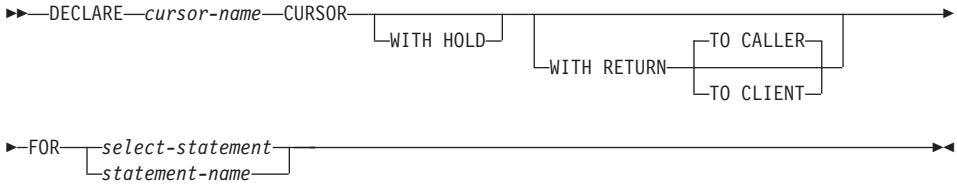
If *statement-name* is specified:

- 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.

If *select-statement* is specified:

- GROUP privileges are not checked.
- The authorization ID of the statement is the authorization ID specified during program preparation.

Syntax:



Description:

cursor-name
Specifies the name of the cursor created when the source program is run. The name must not be the same as the name of another cursor declared in the source program. The cursor must be opened before use.

WITH HOLD
Maintains resources across multiple units of work. The effect of the WITH HOLD cursor attribute is as follows:

- For units of work ending with COMMIT:
  - Open cursors defined WITH HOLD remain open. The cursor is positioned before the next logical row of the results table.

    If a DISCONNECT statement is issued after a COMMIT statement for a connection with WITH HOLD cursors, the held cursors must be explicitly closed or the connection will be assumed to have performed work (simply by having open WITH HELD cursors even though no SQL statements were issued) and the DISCONNECT statement will fail.
  - All locks are released, except locks protecting the current cursor position of open WITH HOLD cursors. The locks held include the locks on the table, and for parallel environments, the locks on rows where the cursors are currently positioned. Locks on packages and dynamic SQL sections (if any) are held.
  - Valid operations on cursors defined WITH HOLD immediately following a COMMIT request are:
    - FETCH: Fetches the next row of the cursor.
    - CLOSE: Closes the cursor.
  - UPDATE and DELETE CURRENT OF CURSOR are valid only for rows that are fetched within the same unit of work.
  - LOB locators are freed.
- For units of work ending with ROLLBACK:
  - All open cursors are closed.
  - All locks acquired during the unit of work are released.
  - LOB locators are freed.
- For special COMMIT case:

– Packages may be recreated either explicitly, by binding the package, or implicitly, because the package has been invalidated and then dynamically recreated the first time it is referenced. All held cursors are closed during package rebind. This may result in errors during subsequent execution.

WITH RETURN
This clause indicates that the cursor is intended for use as a result set from a stored procedure. WITH RETURN is relevant only if the DECLARE CURSOR statement is contained with the source code for a stored procedure. In other cases, the precompiler may accept the clause, but it has no effect.

Within an SQL procedure, cursors declared using the WITH RETURN clause that are still open when the SQL procedure ends, define the result sets from the SQL procedure. All other open cursors in an SQL procedure are closed when the SQL procedure ends. Within an external stored procedure (one not defined using LANGUAGE SQL), the default for all cursors is WITH RETURN TO CALLER. Therefore, all cursors that are open when the procedure ends will be considered result sets.

TO CALLER
Specifies that the cursor can return a result set to the caller. For example, if the caller is another stored procedure, the result set is returned to that stored procedure. If the caller is a client application, the result set is returned to the client application.

TO CLIENT
Specifies that the cursor can return a result set to the client application. This cursor is invisible to any intermediate nested procedures. If a function or method called the procedure either directly or indirectly, result sets cannot be returned to the client and the cursor will be closed after the procedure finishes.

select-statement
Identifies the SELECT statement of the cursor. The *select-statement* must not include parameter markers, but can include references to host variables. The declarations of the host variables must precede the DECLARE CURSOR statement in the source program.

statement-name
The SELECT statement of the cursor is the prepared SELECT statement identified by the *statement-name* when the cursor is opened. The *statement-name* must not be identical to a *statement-name* specified in another DECLARE CURSOR statement of the source program.

For an explanation of prepared SELECT statements, see “PREPARE”.

Notes:

- A program called from another program, or from a different source file within the same program, cannot use the cursor that was opened by the calling program.
- Unnested stored procedures, with LANGUAGE other than SQL, will have WITH RETURN TO CALLER as the default behavior if DECLARE CURSOR is specified without a WITH RETURN clause, and the cursor is left open in the procedure. This provides compatibility with stored procedures from previous versions that allow stored procedures to return result sets to applicable client applications. To avoid this behavior, close all cursors opened in the procedure.
- 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. This value is determined when the cursor is opened.
- For more efficient processing of data, the database manager can block data for read-only cursors when retrieving data from a remote server. The use of the FOR UPDATE clause helps the database manager decide whether a cursor is updatable or not. Updatability is also used to determine the access path selection as well. If a cursor is not going to be used in a Positioned UPDATE or DELETE statement, it should be declared as FOR READ ONLY.
- 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:
  - each FROM clause of the outer fullselect identifies only one base table or deletable view (cannot identify a nested or common table expression or a nickname) without use of the OUTER clause
  - the outer fullselect does not include a VALUES clause
  - the outer fullselect does not include a GROUP BY clause or HAVING clause
  - the outer fullselect does not include column functions in the select list
  - the outer fullselect does not include SET operations (UNION, EXCEPT, or INTERSECT) with the exception of UNION ALL
  - the select list 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 FOR READ ONLY clause
  - one or more of the following is true:
    - the FOR UPDATE clause is specified
    - the cursor is statically defined
    - the LANGLEVEL bind option is MIA or SQL92E

A column in the select list of the outer fullselect associated with a cursor is *updatable* if all of the following are true:

- the cursor is deletable
- the column resolves to a column of the base table
- the LANGLEVEL bind option is MIA, SQL92E or the select-statement includes the FOR UPDATE clause (the column must be specified explicitly or implicitly in the FOR UPDATE clause).

A cursor is *read-only* if it is not deletable.

A cursor is *ambiguous* if all of the following are true:

- the select-statement is dynamically prepared
- the select-statement does not include either the FOR READ ONLY clause or the FOR UPDATE clause
- the LANGLEVEL bind option is SAA1
- the cursor otherwise satisfies the conditions of a deletable cursor.

An ambiguous cursor is considered read-only if the BLOCKING bind option is ALL, otherwise it is considered updatable.

- Cursors in stored procedures that are called by application programs written using CLI can be used to define result sets that are returned directly to the client application. Cursors in SQL procedures can also be returned to a calling SQL procedure only if they are defined using the WITH RETURN clause.
- Cursors declared in routines that are invoked directly or indirectly from a cursor declared WITH HOLD, do not inherit the WITH HOLD option. Thus, unless the cursor in the routine is explicitly defined WITH HOLD, a COMMIT in the application will close it.

  Consider the following application and two UDFs:

```
Application:

   DECLARE APPCUR CURSOR WITH HOLD FOR SELECT UDF1() ...
   OPEN APPCUR
   FETCH APPCUR ...
   COMMIT

UDF1:

   DECLARE UDF1CUR CURSOR FOR SELECT UDF2() ...
   OPEN UDF1CUR
   FETCH UDF1CUR ...

UDF2:
```

```
DECLARE UDF2CUR CURSOR WITH HOLD FOR SELECT UDF2() ...
OPEN UDF2CUR
FETCH UDF2CUR ...
```

After the application fetches cursor APPCUR, all three cursors are open. When the application issues the COMMIT statement, APPCUR remains open, because it was declared WITH HOLD. In UDF1, however, the cursor UDF1CUR is closed, because it was not defined with the WITH HOLD option. When the cursor UDF1CUR is closed, all routine invocations in the corresponding select-statement complete (receiving a final call, if so defined). UDF2 completes, which causes UDF2CUR to close.

Example:

The DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DEPARTMENT
  WHERE ADMRDEPT = 'A00';
```

Related reference:

- "Select-statement" in the *SQL Reference, Volume 1*
- "CALL" on page 101
- "OPEN" on page 615
- "PREPARE" on page 620

Related samples:

- "cursor.sqb -- How to update table data with cursor statically (MF COBOL)"
- "tabsql.sqb -- Demonstrates common table expressions using SQL (MF COBOL)"
- "fnuse.sqc -- How to use built-in SQL functions (C)"
- "tbinfo.sqc -- How to get information at the table level (C)"
- "tut_mod.sqc -- How to modify table data (C)"
- "tut_read.sqc -- How to read tables (C)"
- "fnuse.sqC -- How to use built-in SQL functions (C++)"
- "tbinfo.sqC -- How to get information at the table level (C++)"
- "tut_mod.sqC -- How to modify table data (C++)"
- "tut_read.sqC -- How to read tables (C++)"

DECLARE GLOBAL TEMPORARY TABLE

The DECLARE GLOBAL TEMPORARY TABLE statement defines a temporary table for the current session. The declared temporary table description does not appear in the system catalog. It is not persistent and cannot be shared with other sessions. Each session that defines a declared global temporary table of the same name has its own unique description of the temporary table. When the session terminates, the rows of the table are deleted, and the description of the temporary table is dropped.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

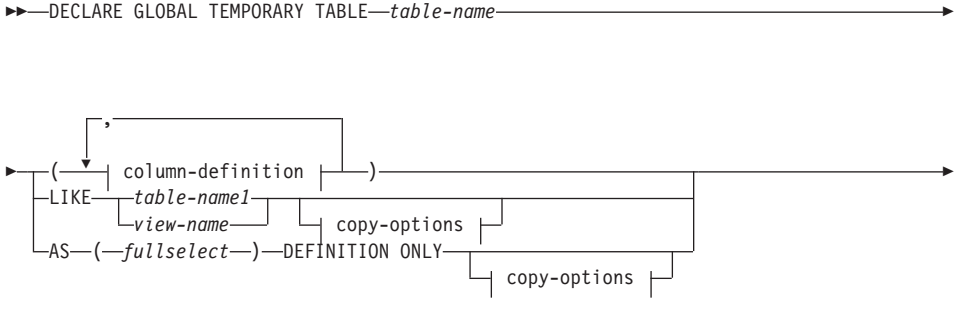
The privileges held by the authorization ID of the statement must include at least one of the following:

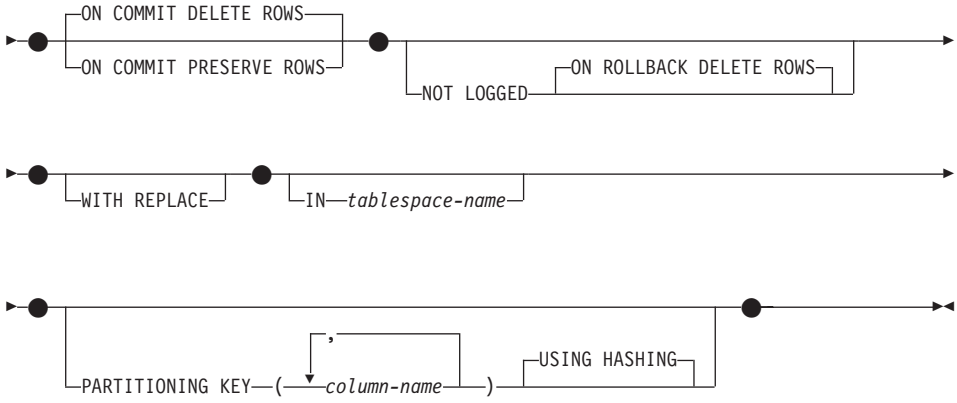
- SYSADM or DBADM authority
- USE privilege on the USER TEMPORARY table space.

When defining a table using LIKE or a fullselect, the privileges held by the authorization ID of the statement must also include at least one of the following on each identified table or view:

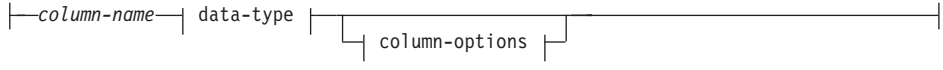
- SELECT privilege on the table or view
- CONTROL privilege on the table or view
- SYSADM or DBADM authority

Syntax:

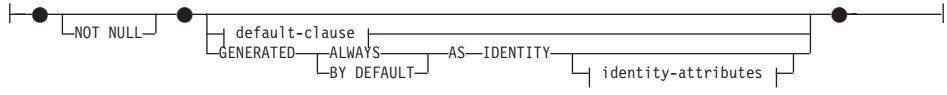




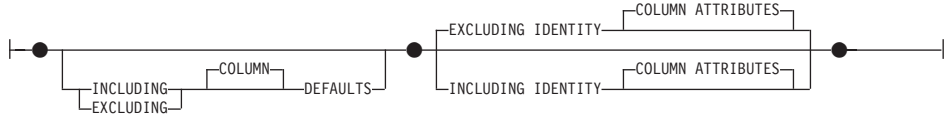
column-definition:



column-options:



copy-options:



Description:

table-name

Names the temporary table. The qualifier, if specified explicitly, must be SESSION, otherwise an error is returned (SQLSTATE 428EK). If the qualifier is not specified, SESSION is implicitly assigned.

Each session that defines a declared global temporary table with the same *table-name* has its own unique description of that declared global temporary table. The WITH REPLACE clause must be specified if *table-name* identifies a declared temporary table that already exists in the session (SQLSTATE 42710).

It is possible that a table, view, alias, or nickname already exists in the catalog, with the same name and the schema name SESSION. In this case:

- A declared global temporary table *table-name* may still be defined without any error or warning
- Any references to SESSION.*table-name* will resolve to the declared global temporary table rather than the SESSION.*table-name* already defined in the catalog.

column-definition
Defines the attributes of a column of the temporary table.

column-name
Names a column of the table. The name cannot be qualified, and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

A table may have the following:

- A 4K page size with a maximum of 500 columns, where the byte counts of the columns must not be greater than 4 005.
- An 8K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 8 101.
- A 16K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 16 293.
- A 32K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 32 677.

For more details, see "Row Size" in "CREATE TABLE".

data-type
For allowable types, see *data-type* in "CREATE TABLE". Note that BLOB, CLOB, DBCLOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, reference, and structured types cannot be used with declared global temporary tables (SQLSTATE 42962). This exception includes distinct types sourced on these restricted types.

FOR BIT DATA can be specified as part of character string data types.

column-options
Defines additional options related to the columns of the table.

NOT NULL
Prevents the column from containing null values. For specification of null values, see NOT NULL in "CREATE TABLE".

default-clause
For specification of defaults, see *default-clause* in "CREATE TABLE".

IDENTITY and identity-attributes
For specification of identity columns, see IDENTITY and *identity-attributes* in "CREATE TABLE".

LIKE *table-name1* 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-name1*) or view (*view-name*), or nickname (*nickname*). The name specified after LIKE must identify a table, view or nickname that exists in the catalog or a declared temporary table. A typed table or typed view cannot be specified (SQLSTATE 428EC).

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table or view.

- If a table is identified, then the implicit definition includes the column name, data type and nullability characteristic of each of the columns of *table-name1*. If EXCLUDING COLUMN DEFAULTS is not specified, then the column default is also included.
- If a view is identified, then the implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of the fullselect defined in *view-name*.

Column default and identity column attributes may be included or excluded, based on the copy-attributes clauses.

The implicit definition does not include any other attributes of the identified table or view. Thus, the new table does not have any unique constraints, foreign key constraints, triggers, or indexes. The table is created in the table space either implicitly or explicitly, as specified by the IN clause.

The names used for *table-name1* and *view-name* can not be the same as the name of the global temporary table that is being created (SQLSTATE 428EC).

AS (*fullselect***) DEFINITION ONLY**
Specifies that the table definition is based on the column definitions from the result of a query expression. The use of AS (*fullselect*) is an implicit definition of *n* columns for the declared global temporary table, where *n* is the number of columns that would result from *fullselect*. The columns of the new table are defined by the columns that result from the *fullselect*. Every select list element must have a unique name (SQLSTATE 42711). The AS clause can be used in the select-clause to provide unique names.

The implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of *fullselect*.

copy-options
These options specify whether or not to copy additional attributes of the source result table definition (table, view, or fullselect).

INCLUDING COLUMN DEFAULTS
Column defaults for each updatable column of the source result table definition are copied. Columns that are not updatable will not have a default defined in the corresponding column of the created table.

If LIKE *table-name1* is specified, and *table-name1* identifies a base table or declared temporary table, then INCLUDING COLUMN DEFAULTS is the default.

EXCLUDING COLUMN DEFAULTS
Column defaults are not copied from the source result table definition.

This clause is the default, except when LIKE *table-name* is specified and *table-name* identifies a base table or declared temporary table.

INCLUDING IDENTITY COLUMN ATTRIBUTES
If available, identity column attributes (START WITH, INCREMENT BY, and CACHE values) are copied from the source's result table definition. It is possible to copy these attributes if the element of the corresponding column in the table, view, or fullselect is the name of a column of a table, or the name of a column of a view, which directly or indirectly maps to the column name of a base table with the identity property. In all other cases, the columns of the new temporary table will not get the identity property. For example:

- 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, except, or intersect).

EXCLUDING IDENTITY COLUMN ATTRIBUTES
Identity column attributes are not copied from the source result table definition.

ON COMMIT
Specifies the action taken on the global temporary table when a COMMIT operation is performed.

DELETE ROWS
All rows of the table will be deleted if no WITH HOLD cursor is open on the table. This is the default.

PRESERVE ROWS
Rows of the table will be preserved.

NOT LOGGED
Changes to the table (including creation of the table) are not logged. If,

when a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed, the table was created in the unit of work (or savepoint), it will be dropped. If the table was dropped in the unit of work (or savepoint), the table will be restored, but with no rows.

ON ROLLBACK DELETE ROWS
Specifies the action that is to be taken on the not logged global temporary table when a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed. If the table data has been changed, all the rows will be deleted.

WITH REPLACE
Indicates that, in the case that a declared global temporary table already exists with the specified name, the existing table is replaced with the temporary table defined by this statement (and all rows of the existing table are deleted).

When WITH REPLACE is not specified, then the name specified must not identify a declared global temporary table that already exists in the current session (SQLSTATE 42710).

IN *tablespace-name*
Identifies the table space in which the global temporary table will be instantiated. The table space must exist and be a USER TEMPORARY table space (SQLSTATE 42838), over which the authorization ID of the statement has USE privilege (SQLSTATE 42501). If this clause is not specified, a table space for the table is determined by choosing the USER TEMPORARY table space with the smallest sufficient page size over which the authorization ID of the statement has USE privilege. When more than one table space qualifies, preference is given according to who was granted the USE privilege:

1. the authorization ID
2. a group to which the authorization ID belongs
3. PUBLIC

If more than one table space still qualifies, the final choice is made by the database manager. When no USER TEMPORARY table space qualifies, an error is raised (SQLSTATE 42727).

Determination of the table space may change when:

- table spaces are dropped or created
- USE privileges are granted or revoked.

The sufficient page size of a table is determined by either the byte count of the row or the number of columns. For more details, see “Row Size” in “CREATE TABLE”.

PARTITIONING KEY (*column-name,...***)**
Specifies the partitioning key used when data in the table is partitioned. Each *column-name* must identify a column of the table and the same column must not be identified more than once.

If this clause is not specified, and this table resides in a multiple partition database partition group, then the partitioning key is defined as the first column of declared temporary table.

For declared temporary tables, in table spaces defined on single-partition database partition groups, any collection of columns can be used to define the partitioning key. If you do not specify this parameter, no partitioning key is created.

USING HASHING
Specifies the use of the hashing function as the partitioning method for data distribution. This is the only partitioning method supported.

Notes:

- ***Compatibilities***
  - For compatibility with DB2 for OS/390 and z/OS:
    - The following syntax is accepted as the default behavior:
      - CCSID ASCII
      - CCSID UNICODE
- **Referencing a declared global temporary table:** The description of a declared global temporary table does not appear in the DB2 catalog (SYSCAT.TABLES); therefore, it is not persistent and is not shareable across database connections. This means that each session that defines a declared global temporary table called *table-name* has its own possibly unique description of that declared global temporary table.

  In order to reference the declared global temporary table in an SQL statement (other than the DECLARE GLOBAL TEMPORARY TABLE statement), the table must be explicitly or implicitly qualified by the schema name SESSION. If *table-name* is not qualified by SESSION, declared global temporary tables are not considered when resolving the reference.

  A reference to SESSION.*table-name* in a connection that has not declared a global temporary table by that name will attempt to resolve from persistent objects in the catalog. If no such object exists, an error occurs (SQLSTATE 42704).
- When binding a package that has static SQL statements that refer to tables implicitly or explicitly qualified by SESSION, those statements will not be bound statically. When these statements are invoked, they will be incrementally bound, regardless of the VALIDATE option chosen while binding the package. At runtime, each table reference will be resolved to a

declared temporary table, if it exists, or a permanent table. If neither exists, an error will be raised (SQLSTATE 42704).

- **Privileges:** When a declared global temporary table is defined, the definer of the table is granted all table privileges on the table, including the ability to drop the table. Additionally, these privileges are granted to PUBLIC. (None of the privileges are granted with the GRANT option, and none of the privileges appear in the catalog table.) This enables any SQL statement in the session to reference a declared global temporary table that has already been defined in that session.
- **Instantiation and Termination:** For the explanations below, P denotes a session and T is a declared global temporary table in the session P:
  - An empty instance of T is created as a result of the DECLARE GLOBAL TEMPORARY TABLE statement that is 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 the SQL procedure compound statement (defined by BEGIN and END), the scope of the declared global temporary table is the connection, not just the compound statement, and the table is known outside of the compound statement. The table is not implicitly dropped at the END of the compound statement. A declared global temporary table cannot be defined multiple times by the same name in other compound statements in that session, unless the table has been explicitly 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, the commit includes the operation DELETE FROM SESSION.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 rollback includes the operation DELETE from SESSION.T.

    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 includes the operation DROP SESSION.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, but the table will have been emptied.
  - When the application process that declared T terminates or disconnects from the database, T is dropped and its instantiated rows are destroyed.
  - When the connection to the server at which T was declared terminates, T is dropped and its instantiated rows are destroyed.

- **Restrictions on the Use of Declared Global Temporary Tables:** Declared Global Temporary tables cannot:
  - Be specified in an ALTER, COMMENT, GRANT, LOCK, RENAME or REVOKE statement (SQLSTATE 42995).
  - Be referenced in a CREATE ALIAS, CREATE FUNCTION (SQL Scalar, Table, or Row), CREATE INDEX, CREATE TRIGGER, or CREATE VIEW statement (SQLSTATE 42995).
  - Be specified in referential constraints (SQLSTATE 42995).

Related reference:

- "CREATE TABLE" on page 332

Related samples:

- "tbtemp.sqc -- How to use a declared temporary table (C)"
- "TbTemp.java -- How to use Declared Temporary Table (JDBC)"

DELETE

The DELETE statement deletes rows from a table, nickname, or view, or executes the delete INSTEAD OF trigger. Deleting a row from a nickname deletes the row from the data source object to which the nickname refers. Deleting a row from a view deletes the row from the table on which the view is based if no INSTEAD OF trigger is defined for the delete operation on this view. If such a trigger is defined, the trigger will be executed 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 DELETE statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

To execute either form of this statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- DELETE privilege on the table, view, or nickname for which rows are to be deleted
- CONTROL privilege on the table, view, or nickname for which rows are to be deleted
- SYSADM or DBADM authority.

To execute a Searched DELETE statement, the privileges held by the authorization ID of the statement must also include at least one of the following for each table, view, or nickname referenced by a subquery:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

If the package used to process the statement is precompiled with SQL92 rules (option LANGLEVEL with a value of SQL92E or MIA), and the searched form of a DELETE statement includes a reference to a column of the table or view in the *search-condition*, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

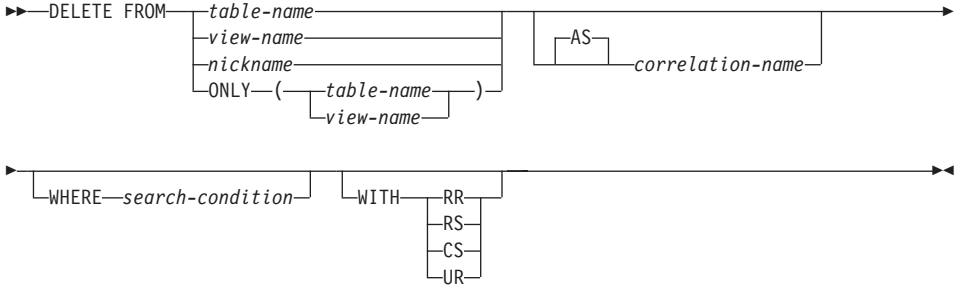
If the specified table or view is preceded by the ONLY keyword, the privileges held by the authorization ID of the statement must also include the SELECT privilege for every subtable or subview of the specified table or view.

Group privileges are not checked for static DELETE statements.

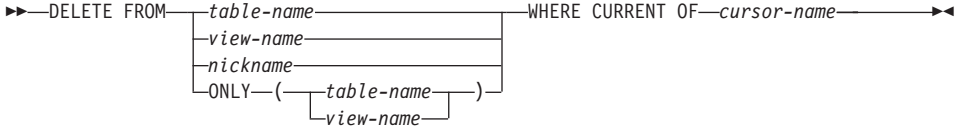
If the target of the delete operation is a nickname, the privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges required for the operation on the object at the data source. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

Syntax:

Searched DELETE:



Positioned DELETE:



Description:

FROM *table-name*, *view-name*, **or** *nickname*
Identifies the table, view, or nickname from which rows are to be deleted. The name must identify a table or view that exists in the catalog, but it must not identify a catalog table, a catalog view, a system-maintained materialized query table, or a read-only view.

If *table-name* is a typed table, rows of the table or any of its proper subtables may get deleted by the statement.

If *view-name* is a typed view, rows of the underlying table or underlying tables of the view's proper subviews may get deleted by the statement. If *view-name* is a regular view with an underlying table that is a typed table, rows of the typed table or any of its proper subtables may get deleted by the statement.

Only the columns of the specified table may be referenced in the WHERE clause. For a positioned DELETE, the associated cursor must also have specified the table or view in the FROM clause without using ONLY.

FROM ONLY (*table-name***)**
Applicable to typed tables, the ONLY keyword specifies that the statement should apply only to data of the specified table and rows of proper subtables cannot be deleted by the statement. For a positioned DELETE, the associated cursor must also have specified the table in the FROM clause using ONLY. If *table-name* is not a typed table, the ONLY keyword has no effect on the statement.

FROM ONLY (*view-name***)**
Applicable to typed views, the ONLY keyword specifies that the statement should apply only to data of the specified view and rows of proper subviews cannot be deleted by the statement. For a positioned DELETE, the associated cursor must also have specified the view in the FROM clause using ONLY. If *view-name* is not a typed view, the ONLY keyword has no effect on the statement.

correlation-name
May be used within the search-condition to designate the table, view, or nickname.

WHERE
Specifies a condition that selects the rows to be deleted. The clause can be omitted, a search condition specified, or a cursor named. If the clause is omitted, all rows of the table or view are deleted.

search-condition
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, view, or nickname, and the deleted rows are those for which the result of the *search-condition* is true.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the *search condition* is applied to a row, and the results used in applying the *search condition*. In actuality, a subquery with no correlated references is executed once,

whereas a subquery with a correlated reference may have to be executed once for each row. If a subquery refers to the object table of a DELETE statement or a dependent table with a delete rule of CASCADE or SET NULL, the subquery is completely evaluated before any rows are deleted.

CURRENT OF *cursor-name*
Identifies a cursor that is defined in a DECLARE CURSOR statement of the program. The DECLARE CURSOR statement must precede the DELETE statement.

The table, view, or nickname named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR".)

When the DELETE statement is executed, the cursor must be positioned on a row: that row is the one deleted. After the deletion, the cursor is positioned before the next row of its result table. If there is no next row, the cursor is positioned after the last row.

WITH
Specifies the isolation level used when locating the rows to be deleted.

RR
Repeatable Read

RS
Read Stability

CS
Cursor Stability

UR
Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound.

Rules:

- ***Triggers:*** DELETE statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the deleted rows. If a DELETE statement on a view causes an INSTEAD OF trigger to fire, referential integrity will be checked against the updates performed in the trigger, and not against the underlying tables of the view that caused the trigger to fire.
- ***Referential Integrity:*** If the identified table or the base table of the identified view is a parent, the rows selected for delete must not have any dependents in a relationship with a delete rule of RESTRICT, and the

DELETE must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT.

If the delete operation is not prevented by a RESTRICT 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 delete rule of NO ACTION is checked to enforce that any non-null foreign key refers to an existing parent row after the other referential constraints have been enforced.

Notes:

- If an error occurs during the execution of a multiple row DELETE, no changes are made to the database.
- Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful DELETE statement. Issuing a COMMIT or ROLLBACK statement will release the locks. 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.
- 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.
- 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. SQLERRD(5) in the SQLCA shows the number of rows affected by referential constraints and by triggered statements. It includes rows that were deleted as a result of a CASCADE delete rule and

rows in which foreign keys were set to NULL as the result of a SET NULL delete rule. With regards to triggered statements, it includes the number of rows that were inserted, updated, or deleted.

- If an error occurs that prevents deleting all rows matching the search condition and all operations required by existing referential constraints, no changes are made to the table and the error is returned.
- For nicknames, the external server option `iud_app_svpt_enforce` poses an additional limitation. Refer to the Federated documentation for more information.
- For some data sources, the SQLCODE -20190 may be returned on a delete against a nickname because of potential data inconsistency. Refer to the Federated documentation for more information.
- For any deleted row that includes currently linked files through DATALINK columns, the files are unlinked, and will be either restored or deleted, depending on the datalink column definition.

  An error may occur when attempting to delete a DATALINK value if the file server referenced in the value is no longer registered with the database server (SQLSTATE 55022).

  An error may also occur when deleting a row that has a link to a server that is unavailable at the time of deletion (SQLSTATE 57050).

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: Delete from the EMPLOYEE table any sales rep or field rep who didn't make a sale in 1995.

```
DELETE FROM EMPLOYEE
  WHERE LASTNAME NOT IN
    (SELECT SALES_PERSON
      FROM SALES
      WHERE YEAR(SALES_DATE)=1995)
      AND JOB IN ('SALESREP','FIELDREP')
```

Related reference:

- "Search conditions" in the *SQL Reference, Volume 1*
- "CREATE VIEW" on page 463

- "DECLARE CURSOR" on page 482
- "SQLCA (SQL communications area)" in the *SQL Reference, Volume 1*

Related samples:

- "dbuse.c -- How to use a database (CLI)"
- "tbmod.c -- How to modify table data (CLI)"
- "dbuse.sqc -- How to use a database (C)"
- "tbconstr.sqc -- How to create, use, and drop constraints (C)"
- "tbmod.sqc -- How to modify table data (C)"
- "dbuse.sqC -- How to use a database (C++)"
- "tbconstr.sqC -- How to create, use, and drop constraints (C++)"
- "tbmod.sqC -- How to modify table data (C++)"
- "delet.sqb -- How to delete table data (MF COBOL)"
- "updat.sqb -- How to update, delete and insert table data (MF COBOL)"
- "DbUse.java -- How to use a database (JDBC)"
- "TbConstr.java -- How to create, use and drop constraints (JDBC)"
- "TbMod.java -- How to modify table data (JDBC)"
- "DbUse.sqlj -- How to use a database (SQLj)"
- "TbConstr.sqlj -- How to create, use and drop constraints (SQLj)"
- "TbMod.sqlj -- How to modify table data (SQLj)"

DESCRIBE

The DESCRIBE statement obtains information about a prepared statement.

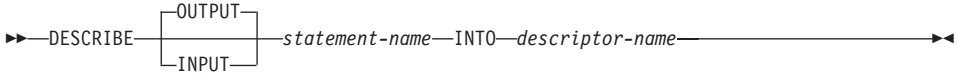
Invocation:

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization:

None required.

Syntax:



Description:

OUTPUT
Optional keyword to indicate that the describe utility should obtain information about the select list columns in the prepared statement, or the output parameter markers associated with the OUT and INOUT parameters, for the stored procedure call.

INPUT
Specifies that the describe utility should obtain information about the input parameter markers in a prepared statement. For a CALL statement, this includes the parameter markers associated with the IN and the INOUT parameters for the stored procedure. Input parameter markers are always considered nullable, regardless of usage.

statement-name
Identifies the statement about which information is required. When the DESCRIBE statement is executed, the name must identify a prepared statement.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA). Before the DESCRIBE statement is executed, the following variables in the SQLDA must be set:

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 DESCRIBE statement is executed.

When the DESCRIBE statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID
The first 6 bytes are set to 'SQLDA ' (that is, 5 letters followed by the space character).

The seventh byte, called SQLDOUBLED, is set to '2' if the SQLDA contains two SQLVAR entries for every select-list item (or, *column* of the result table). This technique is used in order to accommodate LOB, distinct type, structured type, or reference type result columns. Otherwise, SQLDOUBLED is set to the space character.

The doubled flag is set to space if there is not enough room in the SQLDA to contain the entire DESCRIBE reply.

The eighth byte is set to the space character.

SQLDABC
Length of the SQLDA.

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.

If the value is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column of the result table, the second occurrence of SQLVAR contains a description of the second column of the result table, and so on. The description of a column consists of the values assigned to SQLTYPE, SQLLEN, SQLNAME, SQLLONGLEN, and SQLDATATYPE_NAME.

Base SQLVAR

SQLTYPE
A code showing the data type of the column and whether or not it can contain null values.

SQLLEN
A length value depending on the data type of the result columns. SQLLEN is 0 for LOB data types.

SQLNAME
The sqlname is derived as follows:

- If the SQLVAR corresponds to a derived column for a simple column reference in the select list of a select-statement, sqlname is the name of the column.

- If the SQLVAR corresponds to a parameter marker that is not part of an expression in the parameter list of a stored procedure, sqlname contains the name of the parameter if one was specified on CREATE PROCEDURE.
- Otherwise sqlname contains an ASCII numeric literal value that represents the SQLVAR's position within the SQLDA.

Secondary SQLVAR

These variables are only used if the number of SQLVAR entries are doubled to accommodate LOB, distinct type, structured type, or reference type columns.

SQLLONGLEN
The length attribute of a BLOB, CLOB, or DBCLOB column.

SQLDATATYPE_NAME
For any user-defined type (distinct or structured) column, the database manager sets this to the fully qualified user-defined type name. For a reference type column, the database manager sets this to the fully qualified user-defined type name of the target type of the reference. Otherwise, schema name is SYSIBM and the type name is the name in the TYPENAME column of the SYSCAT.DATATYPES catalog view.

Notes:

- Before the DESCRIBE statement is executed, the value of SQLN must be set to indicate how many occurrences of SQLVAR are provided in the SQLDA and enough storage must be allocated to contain SQLN occurrences. For example, to obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR must not be less than the number of columns.
- If a LOB of a large size is expected, then remember that manipulating this large object will affect application memory. Given this condition, consider using locators or file reference variables. Modify the SQLDA after the DESCRIBE statement is executed but prior to allocating storage so that an SQLTYPE of SQL_TYP_xLOB is changed to SQL_TYP_xLOB_LOCATOR or SQL_TYP_xLOB_FILE with corresponding changes to other fields such as SQLLEN. Then allocate storage based on SQLTYPE and continue.
- Code page conversions between extended Unix code (EUC) code pages and DBCS code pages can result in the expansion and contraction of character lengths.
- If a structured type is being selected, but no FROM SQL transform is defined (either because no TRANSFORM GROUP was specified using the

CURRENT DEFAULT TRANSFORM GROUP special register (SQLSTATE 428EM), or because the named group does not have a FROM SQL transform function defined (SQLSTATE 42744)), the DESCRIBE will return an error.

- **Allocating the SQLDA:** Among the possible ways to allocate the SQLDA are the three described below.

  First Technique: Allocate an SQLDA with enough occurrences of SQLVAR to accommodate any select list that the application will have to process. If the table contains any LOB, distinct type, structured type, or reference type columns, the number of SQLVARs should be double the maximum number of columns; otherwise the number should be the same as the maximum number of columns. 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 two steps for every processed select list:

  1. Execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR; that is, an SQLDA for which SQLN is zero. The value returned for SQLD is the number of columns in the result table. This is either the required number of occurrences of SQLVAR or half the required number. Because there were no SQLVAR entries, a warning with SQLSTATE 01005 will be issued. If the SQLCODE accompanying that warning is equal to one of +237, +238 or +239, the number of SQLVAR entries should be double the value returned in SQLD. (The return of these positive SQLCODEs assumes that the SQLWARN bind option setting was YES (return positive SQLCODEs). If SQLWARN was set to NO, +238 is still returned to indicate that the number of SQLVAR entries must be double the value returned in SQLD.)
  2. Allocate an SQLDA with enough occurrences of SQLVAR. Then 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. Execute DESCRIBE and check the SQLD value. Use the SQLD value for the number of occurrences of SQLVAR to allocate a larger SQLDA, if necessary.

  This technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

Example:

In a C program, execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR and then execute a DESCRIBE statement using that SQLDA.

```
EXEC SQL  BEGIN DECLARE SECTION;
  char  stmt1_str[200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

... /* code to prompt user for a query, then to generate */
    /* a select-statement in the stmt1_str           */
EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;

... /* code to set SQLN to zero and to allocate the SQLDA */
EXEC SQL  DESCRIBE STMT1_NAME INTO :sqlda;

... /* code to check that SQLD is greater than zero, to set */
    /* SQLN to SQLD, then to re-allocate the SQLDA          */
EXEC SQL  DESCRIBE STMT1_NAME INTO :sqlda;

... /* code to prepare for the use of the SQLDA             */
    /* and allocate buffers to receive the data             */
EXEC SQL  OPEN DYN_CURSOR;

... /* loop to fetch rows from result table                 */
EXEC SQL  FETCH DYN_CURSOR USING DESCRIPTOR :sqlda;
.
.
.
```

Related reference:

- "PREPARE" on page 620
- "SQLDA (SQL descriptor area)" in the *SQL Reference, Volume 1*

Related samples:

- "tbread.sqc -- How to read tables (C)"
- "tbread.sqC -- How to read tables (C++)"

DISCONNECT

The DISCONNECT statement destroys one or more connections when there is no active unit of work (that is, after a commit or rollback operation). If a single connection is the target of the DISCONNECT statement, the connection is destroyed only if the database has participated in an existing unit of work, regardless of whether there is an active unit of work. For example, if several other databases have done work, but the target in question has not, it can still be disconnected without destroying the connection.

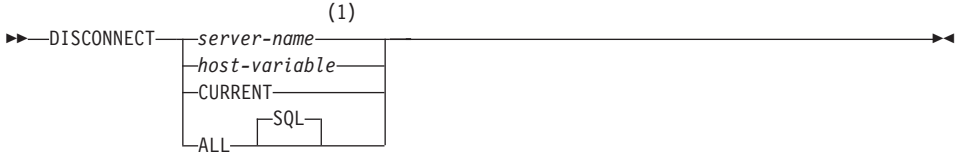
Invocation:

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:

None Required.

Syntax:



Notes:

1 Note that an application server named CURRENT or ALL can only be identified by a host variable.

Description:

server-name or *host-variable*

Identifies the application server by the specified *server-name* or a *host-variable* which contains the *server-name*.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-justified and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The specified database-alias or the database-alias contained in the host variable must identify an existing connection of the application process. If the database-alias does not identify an existing connection, an error (SQLSTATE 08003) is raised.

CURRENT
Identifies the current connection of the application process. The application process must be in the connected state. If not, an error (SQLSTATE 08003) is raised.

ALL
Indicates that all existing connections of the application process are to be destroyed. An error or warning does not occur if no connections exist when the statement is executed. The optional keyword SQL is included to be consistent with the syntax of the RELEASE statement.

Rules:

- Generally, the DISCONNECT statement cannot be executed while within a unit of work. If attempted, an error (SQLSTATE 25000) is raised. The exception to this rule is if a single connection is specified to be disconnected and the database has not participated in an existing unit of work. In this case, it does not matter if there is an active unit of work when the DISCONNECT statement is issued.
- The DISCONNECT statement cannot be executed at all in the Transaction Processing (TP) Monitor environment (SQLSTATE 25000). It is used when the SYNCPOINT precompiler option is set to TWOPHASE.

Notes:

- If the DISCONNECT statement is successful, each identified connection is destroyed.

  If the DISCONNECT statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.
- If DISCONNECT is used to destroy the current connection, the next executed SQL statement should be CONNECT or SET CONNECTION.
- Type 1 CONNECT semantics do not preclude the use of DISCONNECT. However, though DISCONNECT CURRENT and DISCONNECT ALL can be used, they will not result in a commit operation like a CONNECT RESET statement would do.

  If *server-name* or *host-variable* is specified in the DISCONNECT statement, it must identify the current connection because Type 1 CONNECT only supports one connection at a time. Generally, DISCONNECT will fail if within a unit of work with the exception noted in “Rules”.
- Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be destroyed as soon as possible.

- Connections can also be destroyed during a commit operation because the connection option is in effect. The connection option could be AUTOMATIC, CONDITIONAL, or EXPLICIT, which can be set as a precompiler option or through the SET CLIENT API at run time. For information about the specification of the DISCONNECT option, see "Distributed relational databases".

Examples:

Example 1: The SQL connection to IBMSTHDB is no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy the connection.

```
EXEC SQL DISCONNECT IBMSTHDB;
```

Example 2: The current connection is no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy the connection.

```
EXEC SQL DISCONNECT CURRENT;
```

Example 3: The existing connections are no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy all the connections.

```
EXEC SQL DISCONNECT ALL;
```

Related concepts:

- "Distributed relational databases" in the *SQL Reference, Volume 1*

Related samples:

- "dbconn.sqc -- How to connect to and disconnect from a database (C)"
- "dbmcon.sqc -- How to use multiple databases (C)"
- "dbconn.sqC -- How to connect to and disconnect from a database (C++)"
- "dbmcon.sqC -- How to use multiple databases (C++)"
- "Util.java -- Utilities for JDBC sample programs (JDBC)"
- "Util.sqlj -- Utilities for SQLJ sample programs (SQLj)"

DROP

The DROP statement deletes an object. Any objects that are directly or indirectly dependent on that object are either deleted or made inoperative. Whenever an object is deleted, its description is deleted from the catalog and any packages that reference the object are invalidated.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges that must be held by the authorization ID of the DROP statement when dropping objects that allow two-part names must include one of the following or an error will result (SQLSTATE 42501):

- SYSADM or DBADM authority
- DROPIN privilege on the schema for the object
- definer of the object as recorded in the DEFINER column of the catalog view for the object
- CONTROL privilege on the object (applicable only to indexes, index specifications, nicknames, packages, tables, and views).
- definer of the user-defined type as recorded in the DEFINER column of the catalog view SYSCAT.DATATYPES (applicable only when dropping a method associated with a user-defined type)

The authorization ID of the DROP statement when dropping a table or view hierarchy must hold one of the above privileges for each of the tables or views in the hierarchy.

The authorization ID of the DROP statement when dropping a schema must have SYSADM or DBADM authority or be the schema owner as recorded in the OWNER column of SYSCAT.SCHEMATA.

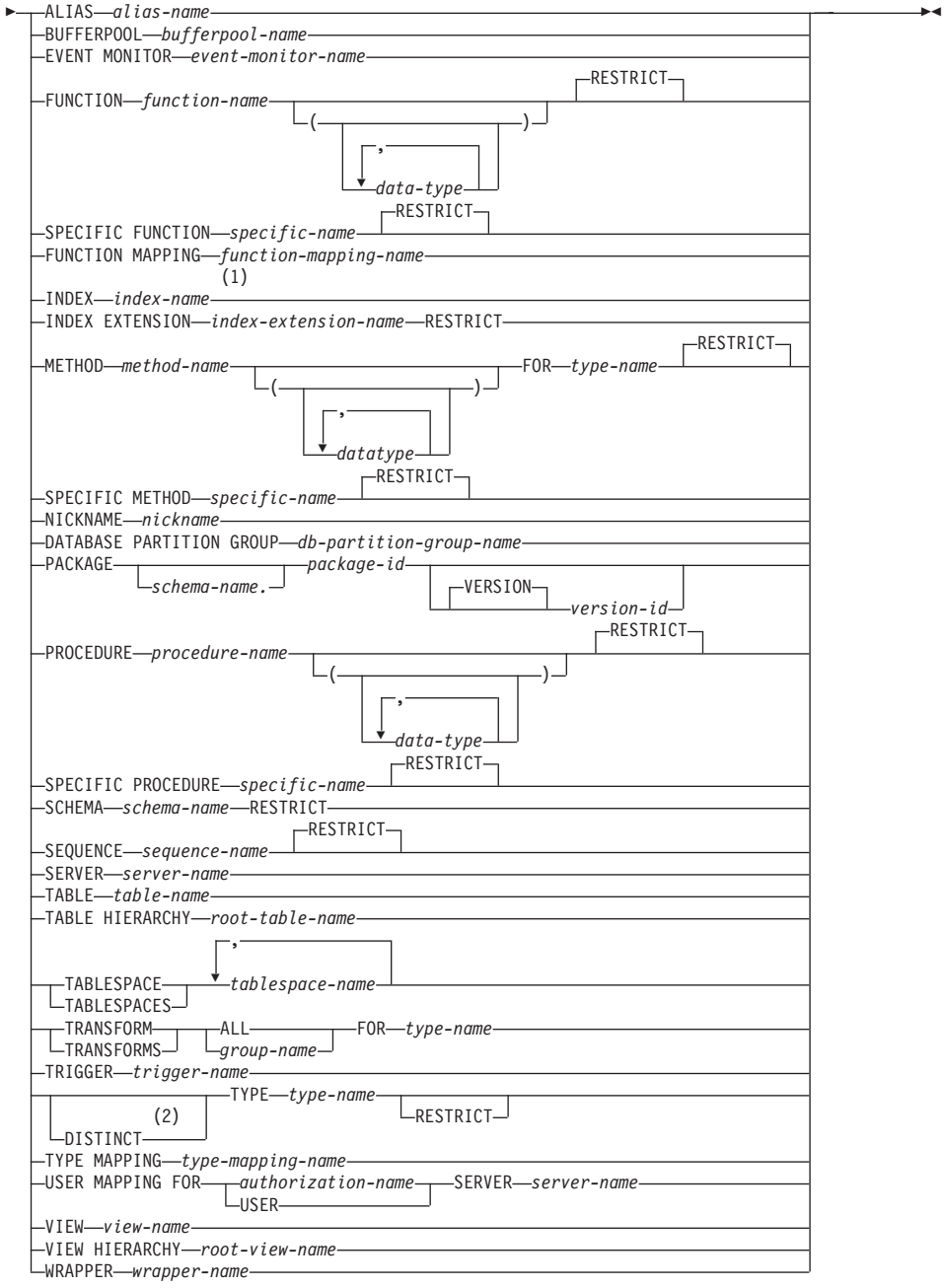
The authorization ID of the DROP statement when dropping a buffer pool, database partition group, or table space must have SYSADM or SYSCTRL authority.

The authorization ID of the DROP statement when dropping an event monitor, server definition, data type mapping, function mapping or a wrapper must have SYSADM or DBADM authority.

The authorization ID of the DROP statement when dropping a user mapping must have SYSADM or DBADM authority, if this authorization ID is different from the federated database authorization name within the mapping. Otherwise, if the authorization ID and the authorization name match, no authorities or privileges are required.

The authorization ID of the DROP statement when dropping a transform must hold SYSADM or DBADM authority, or must be the DEFINER of *type-name*.

Syntax:



Notes:

1 *Index-name* can be the name of either an index or an index specification.

2 DATA can also be used when dropping any user-defined type.

Description:

ALIAS *alias-name*
Identifies the alias that is to be dropped. The *alias-name* must identify an alias that is described in the catalog (SQLSTATE 42704). The specified alias is deleted.

All tables, views, and triggers that reference the alias are made inoperative. (This includes both the table referenced in the ON clause of the CREATE TRIGGER statement, and all tables referenced within the triggered SQL statements.)

BUFFERPOOL *bufferpool-name*
Identifies the buffer pool that is to be dropped. The *bufferpool-name* must identify a buffer pool that is described in the catalog (SQLSTATE 42704). There can be no table spaces assigned to the buffer pool (SQLSTATE 42893). The IBMDEFAULTBP buffer pool cannot be dropped (SQLSTATE 42832). Buffer pool memory is released immediately, to be used by DB2. Disk storage may not be released until the next connection to the database.

EVENT MONITOR *event-monitor-name*
Identifies the event monitor that is to be dropped. The *event-monitor-name* must identify an event monitor that is described in the catalog (SQLSTATE 42704).

If the identified event monitor is ON, an error (SQLSTATE 55034) is returned; otherwise, the event monitor is deleted.

If there are event files in the target path of the event monitor when the event monitor is dropped, the event files are not deleted. However, if a new event monitor that specifies the same target path is created, the event files are deleted.

When dropping WRITE TO TABLE event monitors, table information is removed from the SYSCAT.EVENTTABLES catalog view, but the tables themselves are not dropped.

FUNCTION
Identifies an instance of a user-defined function (either a complete function or a function template) that is to be dropped. The function instance specified must be a user-defined function described in the catalog. Functions implicitly generated by the CREATE DISTINCT TYPE statement cannot be dropped.

There are several different ways available to identify the function instance:

FUNCTION *function-name*
Identifies the particular function, and is valid only if there is exactly one function instance with the *function-name*. The function thus identified may have any number of parameters defined for it. In

dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the function in the named or implied schema, an error (SQLSTATE 42725) is raised.

FUNCTION *function-name* **(***data-type***,...)**
Provides the function signature, which uniquely identifies the function to be dropped. The function selection algorithm is not used.

function-name
Gives the function name of the function to be dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)
Must match the data types that were specified on the CREATE FUNCTION statement in the corresponding position. 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.

If the *data-type* is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead, an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(n) does not need to match the defined value for n since 0<n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

RESTRICT
The RESTRICT keyword enforces the rule that the function is not to be dropped if any of the following dependencies exists:

- Another routine is sourced on the function.

- A view uses the function.
- A trigger uses the function.

RESTRICT is the default behavior.

If no function with the specified signature exists in named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC FUNCTION *specific-name*
Identifies the particular user-defined function that is to be dropped, using the specific name either specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

RESTRICT
The RESTRICT keyword enforces the rule that the function is not to be dropped if any of the following dependencies exists:

- Another routine is sourced on the function.
- A view uses the function.
- A trigger uses the function.

RESTRICT is the default behavior.

It is not possible to drop a function that is in the SYSIBM, SYSFUN, or the SYSPROC schema (SQLSTATE 42832).

Other objects can be dependent upon a function. All such dependencies must be removed before the function can be dropped, with the exception of packages which are marked inoperative. An attempt to drop a function with such dependencies will result in an error (SQLSTATE 42893). See 530 for a list of these dependencies.

If the function can be dropped, it is dropped.

Any package dependent on the specific function being dropped is marked as inoperative. Such a package is not implicitly rebound. It must either be rebound by use of the BIND or REBIND command, or it must be re-prepared by use of the PREP command.

FUNCTION MAPPING *function-mapping-name*
Identifies the function mapping to be dropped. The *function-mapping-name*

must identify a user-defined function mapping that is described in the catalog (SQLSTATE 42704). The function mapping is deleted from the database.

Default function mappings cannot be dropped, but can be disabled.

Packages having a dependency on a dropped function mapping are invalidated.

INDEX *index-name*
Identifies the index or index specification that is to be dropped. The *index-name* must identify an index or index specification that is described in the catalog (SQLSTATE 42704). It cannot be an index required by the system for a primary key or unique constraint or for a replicated materialized query table (SQLSTATE 42917). The specified index or index specification is deleted.

Packages having a dependency on a dropped index or index specification are invalidated.

INDEX EXTENSION *index-extension-name* **RESTRICT**
Identifies the index extension that is to be dropped. The *index-extension-name* must identify an index extension that is described in the catalog (SQLSTATE 42704). The RESTRICT keyword enforces the rule that no index can be defined that depends on this index extension definition (SQLSTATE 42893).

METHOD
Identifies a method body that is to be dropped. The method body specified must be a method described in the catalog (SQLSTATE 42704). Method bodies that are implicitly generated by the CREATE TYPE statement cannot be dropped.

DROP METHOD deletes the body of a method, but the method specification (signature) remains as a part of the definition of the subject type. After dropping the body of a method, the method specification can be removed from the subject type definition by ALTER TYPE DROP METHOD.

There are several ways available to identify the method body to be dropped:

METHOD *method-name*
Identifies the particular method to be dropped, and is valid only if there is exactly one method instance with name *method-name* and subject type *type-name*. Thus, the method identified may have any number of parameters. If no method by this name exists for the type *type-name*, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the method for the named data type, an error (SQLSTATE 42725) is raised.

METHOD *method-name* **(***data-type***,...)**
Provides the method signature, which uniquely identifies the method to be dropped. The method selection algorithm is not used.

method-name
The method name of the method to be dropped for the specified type. The name must be an unqualified identifier.

(*data-type***, ...)**
Must match the data types that were specified in the corresponding positions of the method-specification of the CREATE TYPE or ALTER TYPE statement. The number of data types and the logical concatenation of the data types are used to identify the specific method instance which is to be dropped.

If the data-type is unqualified, the type name is resolved by searching the schemas on the SQL path.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead, an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE TYPE statement.

A type of FLOAT(n) does not need to match the defined value for n since 0<n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no method with the specified signature exists for the named data type, an error is raised (SQLSTATE 42883).

FOR *type-name*
Names the type for which the specified method is to be dropped. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified type name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified type names.

RESTRICT
The RESTRICT keyword enforces the rule that the method is not to be dropped if any of the following dependencies exists:

- Another routine is sourced on the method.
- A view uses the method.
- A trigger uses the method.

RESTRICT is the default behavior.

SPECIFIC METHOD *specific-name*
Identifies the particular method that is to be dropped, using a name either specified or defaulted to at CREATE TYPE or ALTER TYPE time. If the specific name is unqualified, the CURRENT SCHEMA special register is used as a qualifier for an unqualified specific name in dynamic SQL. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for an unqualified specific name. The specific-name must identify a method; otherwise, an error is raised (SQLSTATE 42704).

RESTRICT
The RESTRICT keyword enforces the rule that the method is not to be dropped if any of the following dependencies exists:

- Another routine is sourced on the method.
- A view uses the method.
- A trigger uses the function.

RESTRICT is the default method.

Other objects can be dependent upon a method. All such dependencies must be removed before the method can be dropped, with the exception of packages which will be marked inoperative if the drop is successful. An attempt to drop a method with such dependencies will result in an error (SQLSTATE 42893).

If the method can be dropped, it will be dropped.

Any package dependent on the specific method being dropped is marked as inoperative. Such a package is not implicitly re-bound. Either it must be re-bound by use of the BIND or REBIND command, or it must be re-prepared by use of the PREP command.

If the specific method being dropped overrides another method, all packages dependent on the overridden method — and on methods that override this method in supertypes of the specific method being dropped — are invalidated.

NICKNAME *nickname*
Identifies the nickname that is to be dropped. The nickname must be listed in the catalog (SQLSTATE 42704). The nickname is deleted from the database.

All information about the columns and indexes associated with the nickname is deleted from the catalog. Any materialized query tables that are dependent on the nickname are dropped. Any index specifications that are dependent on the nickname are dropped. Any views that are

dependent on the nickname are marked inoperative. Any packages that are dependent on the dropped index specifications or inoperative views are invalidated. The data source table that the nickname references is not affected.

DATABASE PARTITION GROUP *db-partition-group-name*
Identifies the database partition group that is to be dropped. The *db-partition-group-name* parameter must identify a database partition group that is described in the catalog (SQLSTATE 42704). This is a one-part name.

Dropping a database partition group drops all table spaces defined in the database partition group. All existing database objects with dependencies on the tables in the table spaces (such as packages, referential constraints, etc.) are dropped or invalidated (as appropriate), and dependent views and triggers are made inoperative.

System-defined database partition groups cannot be dropped (SQLSTATE 42832).

If a DROP DATABASE PARTITION GROUP statement is issued against a database partition group that is currently undergoing a data redistribution, the drop database partition group operation fails, and an error is returned (SQLSTATE 55038). However, a partially redistributed database partition group can be dropped. A database partition group can become partially redistributed if a REDISTRIBUTE DATABASE PARTITION GROUP command does not execute to completion. This can happen if it is interrupted by either an error or a FORCE APPLICATION ALL command. (For a partially redistributed database partition group, the REBALANCE_PMAP_ID in the SYSCAT.DBPARTITIONGROUPS catalog is not −1.)

PACKAGE *schema-name.package-id*
Identifies the package that is to be dropped. If a schema name is not specified, the package ID is implicitly qualified by the default schema. The schema name and package ID, together with the implicitly or explicitly specified version ID, must identify a package that is described in the catalog (SQLSTATE 42704). The specified package is deleted. If the package being dropped is the only package identified by *schema-name.package-id* (that is, there are no other versions), all privileges on the package are also deleted.

VERSION *version-id*
Identifies which package version is to be dropped. If a value is not specified, the version defaults to the empty string. If multiple packages with the same package name but different versions exist, only one package version can be dropped in one invocation of the DROP statement.

PROCEDURE
Identifies an instance of a stored procedure that is to be dropped. The procedure instance specified must be a stored procedure described in the catalog.

There are several different ways available to identify the procedure instance:

PROCEDURE *procedure-name*
Identifies the particular procedure to be dropped, and is valid only if there is exactly one procedure instance with the *procedure-name* in the schema. The procedure thus identified may have any number of parameters defined for it. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If there is more than one specific instance of the procedure in the named or implied schema, an error (SQLSTATE 42725) is raised.

RESTRICT
The RESTRICT keyword prevents the procedure from being dropped if a trigger definition contains a CALL statement with the name of the procedure. RESTRICT is the default behavior.

PROCEDURE *procedure-name* **(***data-type***,...)**
Provides the procedure signature, which uniquely identifies the procedure to be dropped. The procedure selection algorithm is not used.

procedure-name
Gives the procedure name of the procedure to be dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)
Must match the data types that were specified on the CREATE PROCEDURE statement in the corresponding position. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be dropped.

If the *data-type* is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead, an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(n) does not need to match the defined value for n since 0<n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in named or implied schema, an error (SQLSTATE 42883) is returned.

SPECIFIC PROCEDURE *specific-name*
Identifies the particular stored procedure that is to be dropped, using the specific name either specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

RESTRICT
The RESTRICT keyword prevents the procedure from being dropped if a trigger definition contains a CALL statement with the name of the procedure. RESTRICT is the default behavior.

It is not possible to drop a procedure that is in the SYSIBM, SYSFUN, or the SYSPROC schema (SQLSTATE 42832).

SCHEMA *schema-name* **RESTRICT**
Identifies the particular schema to be dropped. The *schema-name* must identify a schema that is described in the catalog (SQLSTATE 42704). The RESTRICT keyword enforces the rule that no objects can be defined in the specified schema for the schema to be deleted from the database (SQLSTATE 42893).

SEQUENCE *sequence-name*
Identifies the particular sequence that is to be dropped. The *sequence-name*, along with the implicit or explicit schema name, must identify an existing

sequence at the current server. If no sequence by this name exists in the explicitly or implicitly specified schema, an error (SQLSTATE 42704) is raised.

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 NEXTVAL or PREVVAL expression in the trigger specifies the sequence (SQLSTATE 42893).
- An SQL function or an SQL method exists such that a NEXTVAL expression in the routine body specifies the sequence (SQLSTATE 42893).

SERVER *server-name*
Identifies the data source whose definition is to be dropped from the catalog. The *server-name* must identify a data source that is described in the catalog (SQLSTATE 42704). The definition of the data source is deleted.

All nicknames for tables and views residing at the data source are dropped. Any index specifications dependent on these nicknames are dropped. Any user-defined function mappings, user-defined type mappings, and user mappings that are dependent on the dropped server definition are also dropped. All packages dependent on the dropped server definition, function mappings, nicknames, and index specifications are invalidated.

TABLE *table-name*
Identifies the base table, declared temporary table, materialized query table, or staging table that is to be dropped. The *table-name* must identify a table that is described in the catalog or, if it is a declared temporary table, then the *table-name* must be qualified by the schema name SESSION and exist in the application (SQLSTATE 42704). The subtables of a typed table are dependent on their supertables. All subtables must be dropped before a supertable can be dropped (SQLSTATE 42893). The specified table is deleted from the database.

All indexes, primary keys, foreign keys, check constraints, materialized query tables, and staging tables referencing the table are dropped. All views and triggers that reference the table are made inoperative. (This includes both the table referenced in the ON clause of the CREATE TRIGGER statement, and all tables referenced within the triggered SQL statements.) All packages depending on any object dropped or marked inoperative will be invalidated. This includes packages dependent on any supertables above the subtable in the hierarchy. Any reference columns for which the dropped table is defined as the scope of the reference become unscoped.

Packages are not dependent on declared temporary tables, and therefore are not invalidated when such a table is dropped.

All files that are linked through any DATALINK columns are unlinked. The unlink operation is performed asynchronously so the files may not be immediately available for other operations.

When a subtable is dropped from a table hierarchy, the columns associated with the subtable are no longer accessible although they continue to be considered with respect to limits on the number of columns and size of the row. Dropping a subtable has the effect of deleting all the rows of the subtable from the supertables. This may result in activation of triggers or referential integrity constraints defined on the supertables.

When a declared temporary table is dropped, and its creation preceded the active unit of work or savepoint, then the table will be functionally dropped and the application will not be able to access the table. However, the table will still reserve some space in its table space and will prevent that USER TEMPORARY table space from being dropped or the database partition group of the USER TEMPORARY table space from being redistributed until the unit of work is committed or savepoint is ended. Dropping a declared temporary table causes the data in the table to be destroyed, regardless of whether DROP is committed or rolled back.

A table cannot be dropped if it has the RESTRICT ON DROP attribute.

TABLE HIERARCHY *root-table-name*
Identifies the typed table hierarchy that is to be dropped. The *root-table-name* must identify a typed table that is the root table in the typed table hierarchy (SQLSTATE 428DR). The typed table identified by *root-table-name* and all of its subtables are deleted from the database.

All indexes, materialized query tables, staging tables, primary keys, foreign keys, and check constraints referencing the dropped tables are dropped. All views and triggers that reference the dropped tables are made inoperative. All packages depending on any object dropped or marked inoperative will be invalidated. Any reference columns for which one of the dropped tables is defined as the scope of the reference become unscoped.

All files that are linked through any DATALINK columns are unlinked. The unlink operation is performed asynchronously so the files may not be immediately available for other operations.

Unlike dropping a single subtable, dropping the table hierarchy does not result in the activation of delete triggers of any tables in the hierarchy nor does it log the deleted rows.

TABLESPACE or **TABLESPACES** *tablespace-name*
Identifies the table spaces that are to be dropped. *tablespace-name* must identify a table space that is described in the catalog (SQLSTATE 42704). This is a one-part name.

The table spaces will not be dropped (SQLSTATE 55024) if there is any table that stores at least one of its parts in a table space being dropped, and has one or more of its parts in another table space that is not being dropped (these tables would need to be dropped first), or if any table that resides in the table space has the RESTRICT ON DROP attribute. System table spaces cannot be dropped (SQLSTATE 42832). A SYSTEM TEMPORARY table space cannot be dropped (SQLSTATE 55026) if it is the only temporary table space that exists in the database. A USER TEMPORARY table space cannot be dropped if there is a declared temporary table created in it (SQLSTATE 55039). Even if a declared temporary table has been dropped, the USER TEMPORARY table space will still be considered to be in use until the unit of work containing the DROP TABLE has been committed.

Dropping a table space drops all objects defined in the table space. All existing database objects with dependencies on the table space, such as packages, referential constraints, etc. are dropped or invalidated (as appropriate), and dependent views and triggers are made inoperative.

Containers created by the user are not deleted. Any directories in the path of the container name that were created by the database manager on CREATE TABLESPACE will be deleted. All containers that are below the database directory are deleted. For SMS table spaces, the deletions occur after all connections are disconnected or the DEACTIVATE DATABASE command is issued.

TRANSFORM ALL FOR *type-name*
Indicates that all transforms groups defined for the user-defined data type *type-name* are to be dropped. The transform functions referenced in these groups are not dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *type-name* must identify a user-defined type described in the catalog (SQLSTATE 42704).

If there are not transforms defined for *type-name*, an error is raised (SQLSTATE 42740).

DROP TRANSFORM is the inverse of CREATE TRANSFORM. It causes the transform functions associated with certain groups, for a given datatype, to become undefined. The functions formerly associated with these groups still exist and can still be called explicitly, but they no longer

have the transform property, and are no longer invoked implicitly for exchanging values with the host language environment.

The transform group is not dropped if there is a user-defined function (or method) written in a language other than SQL that has a dependency on one of the group's transform functions defined for the user-defined type *type-name* (SQLSTATE 42893). Such a function has a dependency on the transform function associated with the referenced transform group defined for type *type-name*. Packages that depend on a transform function associated with the named transform group are marked inoperative.

TRANSFORMS *group-name* **FOR** *type-name*
Indicates that the specified transform group for the user-defined data type *type-name* is to be dropped. The transform functions referenced in this group are not dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *type-name* must identify a user-defined type described in the catalog (SQLSTATE 42704), and the *group-name* must identify an existing transform group for *type-name*.

TRIGGER *trigger-name*
Identifies the trigger that is to be dropped. The *trigger-name* must identify a trigger that is described in the catalog (SQLSTATE 42704). The specified trigger is deleted.

Dropping triggers causes certain packages to be marked invalid.

If *trigger-name* specifies an INSTEAD OF trigger on a view, another trigger may depend on that trigger through an update against the view.

TYPE *type-name*
Identifies the user-defined type to be dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. For a structured type, the associated reference type is also dropped. The *type-name* must identify a user-defined type described in the catalog. If DISTINCT is specified, then the *type-name* must identify a distinct type described in the catalog.

RESTRICT
The type is not dropped (SQLSTATE 42893) if any of the following is true:

- The type is used as the type of a column of a table or view.
- The type has a subtype.
- The type is a structured type used as the data type of a typed table or a typed view.

- The type is an attribute of another structured type.
- There exists a column of a table whose type might contain an instance of *type-name*. This can occur if *type-name* is the type of the column or is used elsewhere in the column's associated type hierarchy. More formally, for any type T, T cannot be dropped if there exists a column of a table whose type directly or indirectly uses *type-name*.
- The type is the target type of a reference-type column of a table or view, or a reference-type attribute of another structured type.
- The type or a reference to the type is a parameter type or a return value type of a function or method that cannot be dropped.
- The type, or a reference to the type, is used in the body of an SQL function or method, but it is not a parameter type or a return value type.
- The type is used in a check constraint, trigger, view definition, or index extension.

Functions that use the type: If the user-defined type can be dropped, then for every function, F (with specific name SF), that has parameters or a return value of the type being dropped or a reference to the type being dropped, the following DROP FUNCTION statement is effectively executed:

```
DROP SPECIFIC FUNCTION SF
```

It is possible that this statement also would cascade to drop dependent functions. If all of these functions are also in the list to be dropped because of a dependency on the user-defined type, the drop of the user-defined type will succeed (otherwise it fails with SQLSTATE 42893).

Methods that use the type: If the user-defined type can be dropped, then for every method, M of type T1 (with specific name SM), that has parameters or a return value of the type being dropped or a reference to the type being dropped, the following statements are effectively executed:

```
DROP SPECIFIC METHOD SM
ALTER TYPE T1 DROP SPECIFIC METHOD SM
```

The existence of objects that are dependent on these methods may cause the DROP TYPE operation to fail.

All packages that are dependent on methods defined in supertypes of the type being dropped, and that are eligible for overriding, are invalidated.

Specifying RESTRICT is optional, because RESTRICT reflects the default behavior.

TYPE MAPPING *type-mapping-name*
Identifies the user-defined data type mapping to be dropped. The *type-mapping-name* must identify a data type mapping that is described in the catalog (SQLSTATE 42704). The data type mapping is deleted from the database.

No additional objects are dropped.

USER MAPPING FOR *authorization-name* | **USER SERVER** *server-name*
Identifies the user mapping to be dropped. This mapping associates an authorization name that is used to access the federated database with an authorization name that is used to access a data source. The first of these two authorization names is either identified by the *authorization-name* or referenced by the special register USER. The *server-name* identifies the data source that the second authorization name is used to access.

The *authorization-name* must be listed in the catalog (SQLSTATE 42704). The *server-name* must identify a data source that is described in the catalog (SQLSTATE 42704). The user mapping is deleted.

No additional objects are dropped.

VIEW *view-name*
Identifies the view that is to be dropped. The *view-name* must identify a view that is described in the catalog (SQLSTATE 42704). The subviews of a typed view are dependent on their superviews. All subviews must be dropped before a superview can be dropped (SQLSTATE 42893).

The specified view is deleted. The definition of any view or trigger that is directly or indirectly dependent on that view is marked inoperative. Any materialized query table or staging table that is dependent on any view that is marked inoperative is dropped. Any packages dependent on a view that is dropped or marked inoperative will be invalidated. This includes packages dependent on any superviews above the subview in the hierarchy. Any reference columns for which the dropped view is defined as the scope of the reference become unscoped.

VIEW HIERARCHY *root-view-name*
Identifies the typed view hierarchy that is to be dropped. The *root-view-name* must identify a typed view that is the root view in the typed view hierarchy (SQLSTATE 428DR). The typed view identified by *root-view-name* and all of its subviews are deleted from the database.

The definition of any view or trigger that is directly or indirectly dependent on any of the dropped views is marked inoperative. Any packages dependent on any view or trigger that is dropped or marked

inoperative will be invalidated. Any reference columns for which a dropped view or view marked inoperative is defined as the scope of the reference become unscoped.

WRAPPER *wrapper-name*
Identifies the wrapper to be dropped. The *wrapper-name* must identify a wrapper that is described in the catalog (SQLSTATE 42704). The wrapper is deleted.

All server definitions, user-defined function mappings, and user-defined data type mappings that are dependent on the wrapper are dropped. All user-defined function mappings, nicknames, user-defined data type mappings, and user mappings that are dependent on the dropped server definitions are also dropped. Any index specifications dependent on the dropped nicknames are dropped, and any views dependent on these nicknames are marked inoperative. All packages dependent on the dropped objects and inoperative views are invalidated.

Rules:

Dependencies: Table 11 on page 531 shows the dependencies that objects have on each other. Not all dependencies are explicitly recorded in the catalog. For example, there is no record of the constraints on which a package has dependencies. Four different types of dependencies are shown:

R Restrict semantics. The underlying object cannot be dropped as long as the object that depends on it exists.

C Cascade semantics. Dropping the underlying object causes the object that depends on it (the depending object) to be dropped as well. However, if the depending object cannot be dropped because it has a Restrict dependency on some other object, the drop of the underlying object will fail.

X Inoperative semantics. Dropping the underlying object causes the object that depends on it to become inoperative. It remains inoperative until a user takes some explicit action.

A Automatic Invalidation/Revalidation semantics. Dropping the underlying object causes the object that depends on it to become invalid. The database manager attempts to revalidate the invalid object.

A package used by a function or a method, or by a procedure that is called directly or indirectly from a function or method, will only be automatically revalidated if the routine is defined as MODIFIES SQL DATA. If the routine is not MODIFIES SQL DATA, an error is returned (SQLSTATE 56098).

Some DROP statement parameters and objects are not shown in Table 11 because they would result in blank rows or columns:

- EVENT MONITOR, PACKAGE, PROCEDURE, SCHEMA, TYPE MAPPING, and USER MAPPING DROP statements do not have object dependencies.
- Alias, bufferpool, partitioning key, privilege, and procedure object types do not have DROP statement dependencies.
- A DROP SERVER, DROP FUNCTION MAPPING, or DROP TYPE MAPPING statement in a given unit of work (UOW) cannot be processed under either of the following conditions:
  - The statement references a single data source, and the UOW already includes a SELECT statement that references a nickname for a table or view within this data source (SQLSTATE 55006).
  - The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes a SELECT statement that references a nickname for a table or view within one of these data sources (SQLSTATE 55006).

Table 11. Dependencies

Object Type → / Statement ↓	CONSTRAINT	FUNCTION	FUNCMAPPING	INDEX	INDEXEXTENSION	METHOD	NICKNAME	NODEGROUP	PACKAGE[31]	SERVER	TABLE	TABLESPACE	TRIGGER	TYPE	TYPEMAPPING	USERMAPPING	VIEW
ALTER FUNCTION	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-
ALTER METHOD	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-
ALTER NICKNAME	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-
ALTER PROCEDURE	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-
ALTER SERVER	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-
ALTER TABLE DROP CONSTRAINT	C	-	-	-	-	-	-	-	A[1]	-	-	-	-	-	-	-	-

Table 11. Dependencies (continued)

Object Type → / Statement ↓	CONSTRAINT	FUNCTION	FUNCMAPPING	INDEX	INDEXEXTENSION	METHOD	NICKNAME	NODEGROUP	PACKAGE[31]	SERVER	TABLE	TABLESPACE	TRIGGER	TYPE	TYPEMAPPING	USERMAPPING	VIEW
ALTER TABLE DROP PARTITIONING KEY	-	-	-	-	-	-	-	R[20]	A[1]	-	-	-	-	-	-	-	-
ALTER TYPE ADD ATTRIBUTE	-	-	-	-	R	-	-	-	A[23]	-	R[24]	-	-	-	-	-	R[14]
ALTER TYPE ALTER METHOD	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-
ALTER TYPE DROP ATTRIBUTE	-	-	-	-	R	-	-	-	A[23]	-	R[24]	-	-	-	-	-	R[14]
ALTER TYPE ADD METHOD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER TYPE DROP METHOD	-	-	-	-	-	R[27]	-	-	-	-	-	-	-	-	-	-	-
CREATE METHOD	-	-	-	-	-	-	-	-	A[28]	-	-	-	-	-	-	-	-
CREATE TYPE	-	-	-	-	-	-	-	-	A[29]	-	-	-	-	-	-	-	-
DROP ALIAS	-	R	-	-	-	-	-	-	A[3]	-	R[3]	-	X[3]	-	-	-	X[3]
DROP BUFFERPOOL	-	-	-	-	-	-	-	-	-	-	-	R	-	-	-	-	-
DROP DATABASE PARTITION GROUP	-	-	-	-	-	-	-	-	-	-	-	C	-	-	-	-	-
DROP FUNCTION	R	R[7]	R	-	R	R[7]	-	-	X	-	R	-	R	-	-	-	R
DROP FUNCTION MAPPING	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-
DROP INDEX	R	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	R[17]

Table 11. Dependencies (continued)

Object Type → / Statement ↓	CONSTRAINT	FUNCTION	FUNCMAPPING	INDEX	INDEXEXTENSION	METHOD	NICKNAME	NODEGROUP	PACKAGE[31]	SERVER	TABLE	TABLESPACE	TRIGGER	TYPE	TYPEMAPPING	USERMAPPING	VIEW
DROP INDEX EXTENSION	-	R	-	R	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP METHOD	R	R[7]	R	-	R	R	-	-	X/A[30]	-	R	-	R	-	-	-	R
DROP NICKNAME	-	R	-	C	-	R	-	-	A	-	C[11]	-	-	-	-	-	X[16]
DROP PROCEDURE	-	R[7]	-	-	-	R[7]	-	-	X	-	-	-	-	-	-	-	-
DROP SEQUENCE	-	R	-	-	-	R	-	-	A	-	-	-	R	-	-	-	-
DROP SERVER	-	C[21]	C[19]	-	-	-	C	-	A	-	-	-	-	-	C[19]	C	-
DROP TABLE	C	R	-	C	-	-	-	-	A[9]	-	RC[11]	-	X[16]	-	-	-	X[16]
DROP TABLE HIERARCHY	C	R	-	C	-	-	-	-	A[9]	-	RC[11]	-	X[16]	-	-	-	X[16]
DROP TABLESPACE	-	-	-	C[6]	-	-	-	-	-	-	CR[6]	-	-	-	-	-	-
DROP TRANSFORM	-	R	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-
DROP TRIGGER	-	-	-	-	-	-	-	-	A[1]	-	-	-	X[26]	-	-	-	-
DROP TYPE	R[13]	R[5]	-	-	R	-	-	-	A[12]	-	R[18]	-	R[13]	R[4]	-	-	R[14]
DROP VIEW	-	R	-	-	-	-	-	-	A[2]	-	-	-	X[16]	-	-	-	X[15]
DROP VIEW HIERARCHY	-	R	-	-	-	-	-	-	A[2]	-	-	-	X[16]	-	-	-	X[16]
DROP WRAPPER	-	-	C	-	-	-	-	-	-	C	-	-	-	-	C	-	-
REVOKE a privilege[10]	-	CR[25]	-	-	-	CR[25]	-	-	A[1]	-	CX[8]	-	X	-	-	-	X[8]

1 This dependency is implicit in depending on a table with these constraints, triggers, or a partitioning key.

2 If a package has an INSERT, UPDATE, or DELETE statement acting upon a view, then the package has an insert, update or delete usage

on the underlying base table of the view. In the case of UPDATE, the package has an update usage on each column of the underlying base table that is modified by the UPDATE.

If a package has a statement acting on a typed view, creating or dropping any view in the same view hierarchy will invalidate the package.

[3] If a package, materialized query table, staging table, view, or trigger uses an alias, it becomes dependent both on the alias and the object that the alias references. If the alias is in a chain, a dependency is created on each alias in the chain.

Aliases themselves are not dependent on anything. It is possible for an alias to be defined on an object that does not exist.

[4] A user-defined type T can depend on another user-defined type B, if T:

- names B as the data type of an attribute
- has an attribute of REF(B)
- has B as a supertype.

[5] Dropping a data type cascades to drop the functions and methods that use that data type as a parameter or a result type, and methods defined on the data type. Dropping of these functions and methods will not be prevented by the fact that they depend on each other. However, for functions or methods using the datatype within their bodies, restrict semantics apply.

[6] Dropping a table space or a list of table spaces causes all the tables that are completely contained within the given table space or list to be dropped. However, if a table spans table spaces (indexes or long columns in different table spaces) and those table spaces are not in the list being dropped then the table space(s) cannot be dropped as long as the table exists.

[7] A function can depend on another specific function if the depending function names the base function in a SOURCE clause. A function or method can also depend on another specific function or method if the depending routine is written in SQL and uses the base routine in its body. An external method, or an external function with a structured type parameter or returns type will also depend on one or more transform functions.

[8] Only loss of SELECT privilege will cause a materialized query table to be dropped or a view to become inoperative. If the view that is made inoperative is included in a typed view hierarchy, all of its subviews also become inoperative.

[9] If a package has an INSERT, UPDATE, or DELETE statement acting

on table T, then the package has an insert, update or delete usage on T. In the case of UPDATE, the package has an update usage on each column of T that is modified by the UPDATE.

If a package has a statement acting on a typed table, creating or dropping any table in the same table hierarchy will invalidate the package.

10 Dependencies do not exist at the column level because privileges on columns cannot be revoked individually.

If a package, trigger or view includes the use of OUTER(*Z*) in the FROM clause, there is a dependency on the SELECT privilege on every subtable or subview of *Z*. Similarly, if a package, trigger, or view includes the use of DEREF(*Y*) where *Y* is a reference type with a target table or view *Z*, there is a dependency on the SELECT privilege on every subtable or subview of *Z*.

11 A materialized query table is dependent on the underlying tables or nicknames specified in the fullselect of the table definition.

Cascade semantics apply to dependent materialized query tables.

A subtable is dependent on its supertables up to the root table. A supertable cannot be dropped until all of its subtables are dropped.

12 A package can depend on structured types as a result of using the TYPE predicate or the subtype-treatment expression (TREAT *expression* AS *data-type*). The package has a dependency on the subtypes of each structured type specified in the right side of the TYPE predicate, or the right side of the TREAT expression. Dropping or creating a structured type that alters the subtypes on which the package is dependent causes invalidation.

All packages that are dependent on methods defined in supertypes of the type being dropped, and that are eligible for overriding, are invalidated.

13 A check constraint or trigger is dependent on a type if the type is used anywhere in the constraint or trigger. There is no dependency on the subtypes of a structured type used in a TYPE predicate within a check constraint or trigger.

14 A view is dependent on a type if the type is used anywhere in the view definition (this includes the type of typed view). There is no dependency on the subtypes of a structured type used in a TYPE predicate within a view definition.

15 A subview is dependent on its superview up to the root view. A superview cannot be dropped until all its subviews are dropped. Refer to 16 for additional view dependencies.

[16] A trigger or view is also dependent on the target table or target view of a dereference operation or DEREF function. A trigger or view with a FROM clause that includes OUTER(*Z*) is dependent on all the subtables or subviews of *Z* that existed at the time the trigger or view was created.

[17] A typed view can depend on the existence of a unique index to ensure the uniqueness of the object identifier column.

[18] A table may depend on a user defined data type (distinct or structured) because the type is:

- used as the type of a column
- used as the type of the table
- used as an attribute of the type of the table
- used as the target type of a reference type that is the type of a column of the table or an attribute of the type of the table
- directly or indirectly used by a type that is the column of the table.

[19] Dropping a server cascades to drop the function mappings and type mappings created for that named server.

[20] If the partitioning key is defined on a table in a multiple partition database partition group, the partitioning key is required.

[21] If a dependent OLE DB table function has "R" dependent objects (see DROP FUNCTION), then the server cannot be dropped.

[22] An SQL function or method can depend on the objects referenced by its body.

[23] When an attribute A of type TA of *type-name* T is dropped, the following DROP statements are effectively executed:

```
Mutator method: DROP METHOD A (TA) FOR T
Observer method: DROP METHOD A () FOR T
ALTER TYPE T
  DROP METHOD A(TA)
  DROP METHOD A()
```

[24] A table may depend on an attribute of a user-defined structured data type in the following cases:

1. The table is a typed table that is based on *type-name* or any of its subtypes.
2. The table has an existing column of a type that directly or indirectly refers to *type-name*.

[25] A REVOKE of SELECT privilege on a table or view that is used in the body of an SQL function or method body causes an attempt to drop the function or method body, if the function or method body defined no longer has the SELECT privilege. If such a function or method

body is used in a view, trigger, function, or method body, it cannot be dropped, and the REVOKE is restricted as a result. Otherwise, the REVOKE cascades and drops such functions.

[26] A trigger depends on an INSTEAD OF trigger when it modifies the view on which the INSTEAD OF trigger is defined, and the INSTEAD OF trigger fires.

[27] A method declaration of an original method that is overridden by other methods cannot be dropped.(SQLSTATE -2).

[28] If the method of the method body being created is declared to override another method, all packages dependent on the overridden method, and on methods that override this method in supertypes of the method being created, are invalidated.

[29] When a new subtype of an existing type is created, all packages dependent on methods that are defined in supertypes of the type being created, and that are eligible for overriding (for example, no mutators or observers), are invalidated.

[30] If the specific method of the method body being dropped is declared to override another method, all packages dependent on the overridden method, and on methods that override this method in supertypes of the specific method being dropped, are invalidated.

[31] Cached dynamic SQL has the same semantics as packages.

Notes:

- *Compatibilities*
  - For compatibility with previous versions of DB2:
    - NODEGROUP can be specified in place of DATABASE PARTITION GROUP
  - For compatibility with DB2 UDB for OS/390 and z/OS:
    - SYNONYM can be specified in place of ALIAS
    - PROGRAM can be specified in place of PACKAGE
- It is valid to drop a user-defined function while it is in use. Also, a cursor can be open over a statement which contains a reference to a user-defined function, and while this cursor is open the function can be dropped without causing the cursor fetches to fail.
- If a package which depends on a user-defined function is executing, it is not possible for another authorization ID to drop the function until the package completes its current unit of work. At that point, the function is dropped and the package becomes inoperative. The next request for this package results in an error indicating that the package must be explicitly rebound.

- The removal of a function body (this is very different from dropping the function) can occur while an application which needs the function body is executing. This may or may not cause the statement to fail, depending on whether the function body still needs to be loaded into storage by the database manager on behalf of the statement.
- For any dropped table that includes currently linked files through DATALINK columns, the files are unlinked, and will be either restored or deleted, depending on the datalink column definition.
- If a table containing a DATALINK column is dropped while any DB2 Data Links Managers configured to the database are unavailable, either through DROP TABLE or DROP TABLESPACE, then the operation will fail (SQLSTATE 57050).
- In addition to the dependencies recorded for any explicitly specified UDF, the following dependencies are recorded when transforms are implicitly required:
  1. When the structured type parameter or result of a function or method requires a transform, a dependency is recorded for the function or method on the required TO SQL or FROM SQL transform function.
  2. When an SQL statement included in a package requires a transform function, a dependency is recorded for the package on the designated TO SQL or FROM SQL transform function.

  Since the above describes the only circumstances under which dependencies are recorded due to implicit invocation of transforms, no objects other than functions, methods, or packages can have a dependency on implicitly invoked transform functions. On the other hand, explicit calls to transform functions (in views and triggers, for example) do result in the usual dependencies of these other types of objects on transform functions. As a result, a DROP TRANSFORM statement may also fail due to these ″explicit″ type dependencies of objects on the transform(s) being dropped (SQLSTATE 42893).
- Since the dependency catalogs do not distinguish between depending on a function as a transform versus depending on a function by explicit function call, it is suggested that explicit calls to transform functions are not written. In such an instance, the transform property on the function cannot be dropped, or packages will be marked inoperative, simply because they contain explicit invocations in an SQL expression.
- System created sequences for IDENTITY columns cannot be dropped using the DROP SEQUENCE statement.
- When a sequence is dropped, all privileges on the sequence are also dropped and any packages that refer to the sequence are invalidated.

Examples:

Example 1: Drop table TDEPT.

```
DROP TABLE TDEPT
```

Example 2: Drop the view VDEPT.

```
DROP VIEW VDEPT
```

Example 3: The authorization ID HEDGES attempts to drop an alias.

```
DROP ALIAS A1
```

The alias HEDGES.A1 is removed from the catalogs.

Example 4: Hedges attempts to drop an alias, but specifies T1 as the alias-name, where T1 is the name of an existing table (not the name of an alias).

```
DROP ALIAS T1
```

This statement fails (SQLSTATE 42809).

Example 5:

Drop the BUSINESS_OPS database partition group. To drop the database partition group, the two table spaces (ACCOUNTING and PLANS) in the database partition group must first be dropped.

```
DROP TABLESPACE ACCOUNTING
DROP TABLESPACE PLANS
DROP DATABASE PARTITION GROUP BUSINESS_OPS
```

Example 6: Pellow wants to drop the CENTRE function, which he created in his PELLOW schema, using the signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTRE (INT,FLOAT)
```

Example 7: McBride wants to drop the FOCUS92 function, which she created in the PELLOW schema, using the specific name to identify the function instance to be dropped.

```
DROP SPECIFIC FUNCTION PELLOW.FOCUS92
```

Example 8: Drop the function ATOMIC_WEIGHT from the CHEM schema, where it is known that there is only one function with that name.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT
```

Example 9: Drop the trigger SALARY_BONUS, which caused employees under a specified condition to receive a bonus to their salary.

```
DROP TRIGGER SALARY_BONUS
```

Example 10: Drop the distinct data type named shoesize, if it is not currently in use.

```
DROP DISTINCT TYPE SHOESIZE
```

Example 11: Drop the SMITHPAY event monitor.

```
DROP EVENT MONITOR SMITHPAY
```

Example 12: Drop the schema from Example 2 under CREATE SCHEMA using RESTRICT. Notice that the table called PART must be dropped first.

```
  DROP TABLE PART
DROP SCHEMA INVENTRY RESTRICT
```

Example 13: Macdonald wants to drop the DESTROY procedure, which he created in the EIGLER schema, using the specific name to identify the procedure instance to be dropped.

```
DROP SPECIFIC PROCEDURE  EIGLER.DESTROY
```

Example 14: Drop the procedure OSMOSIS from the BIOLOGY schema, where it is known that there is only one procedure with that name.

```
DROP PROCEDURE BIOLOGY.OSMOSIS
```

Example 15: User SHAWN used one authorization ID to access the federated database and another to access the database at an Oracle data source called ORACLE1. A mapping was created between the two authorizations, but SHAWN no longer needs to access the data source. Drop the mapping.

```
DROP USER MAPPING FOR  SHAWN SERVER ORACLE1
```

Example 16: An index of a data source table that a nickname references has been deleted. Drop the index specification that was created to let the optimizer know about this index.

```
DROP INDEX INDEXSPEC
```

Example 17: Drop the MYSTRUCT1 transform group.

```
DROP TRANSFORM MYSTRUCT1 FOR POLYGON
```

Example 18: Drop the method BONUS for the EMP data type in the PERSONNEL schema.

```
DROP METHOD BONUS (SALARY DECIMAL(10,2)) FOR PERSONNEL.EMP
```

Example 19: Drop the sequence ORG_SEQ, with restrictions.

```
DROP SEQUENCE ORG_SEQ
```

Related reference:

- "CREATE TRIGGER" on page 414

- "CREATE VIEW" on page 463
- "CREATE FUNCTION MAPPING" on page 263

Related samples:

- "dbstat.sqb -- Reorganize table and run statistics (MF COBOL)"
- "dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)"
- "tbconstr.sqC -- How to create, use, and drop constraints (C++)"
- "tbcreate.sqC -- How to create and drop tables (C++)"
- "tbtrig.sqC -- How to use a trigger on a table (C++)"
- "DbSeq.java -- How to create, alter and drop a sequence in a database (JDBC)"
- "TbConstr.java -- How to create, use and drop constraints (JDBC)"
- "TbCreate.java -- How to create and drop tables (JDBC)"
- "TbTemp.java -- How to use Declared Temporary Table (JDBC)"
- "TbTrig.java -- How to use triggers (JDBC)"
- "UDFDrop.db2 -- How to uncatalog the Java UDFs contained in UDFsrv.java "
- "spdrop.db2 -- How to uncatalog the stored procedures contained in spserver.sqc "
- "tbconstr.sqc -- How to create, use, and drop constraints (C)"
- "tbcreate.sqc -- How to create and drop tables (C)"
- "tbtemp.sqc -- How to use a declared temporary table (C)"
- "tbtrig.sqc -- How to use a trigger on a table (C)"
- "TbConstr.sqlj -- How to create, use and drop constraints (SQLj)"
- "TbCreate.sqlj -- How to create and drop tables (SQLj)"
- "TbTrig.sqlj -- How to use triggers (SQLj)"

END DECLARE SECTION

The END DECLARE SECTION statement marks the end of a host variable 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 REXX.

Authorization:

None required.

Syntax:

Description:
The END DECLARE SECTION statement can be coded in the application program wherever declarations can appear according to the rules of the host language. It indicates the end of a host variable declaration section. A host variable section starts with a BEGIN DECLARE SECTION statement.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.

Host variable declarations can be specified by using the SQL INCLUDE statement. Otherwise, a host variable declaration section must not contain any statements other than host variable declarations.

Host variables referenced in SQL statements must be declared in a host variable declare section in all host languages, other than REXX. Furthermore, the declaration of each variable must appear before the first reference to the variable.

Variables declared outside a declare section should not have the same name as variables declared within a declare section.

Related reference:

- "BEGIN DECLARE SECTION" on page 98

Related samples:

- "advsql.sqb -- How to read table data using CASE (MF COBOL)"

- “prepbind.sqb -- Precompile and bind an embedded SQL program to a database (MF COBOL)”
- “dtlob.sqc -- How to use the LOB data type (C)”
- “spclient.sqc -- Call various stored procedures (C)”
- “tut_read.sqc -- How to read tables (C)”
- “dtlob.sqC -- How to use the LOB data type (C++)”
- “spclient.sqC -- Call various stored procedures (C++)”
- “tut_read.sqC -- How to read tables (C++)”

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

Invocation:

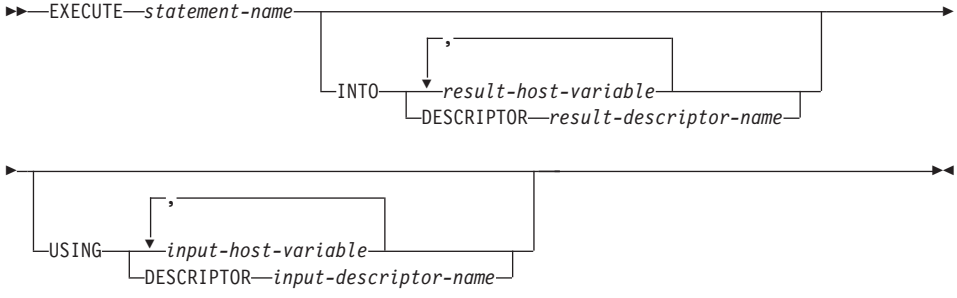
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization:

For statements where authorization checking is performed at statement execution time (DDL, GRANT, and REVOKE statements), the privileges held by the authorization ID of the statement must include those required to execute the SQL statement specified by the PREPARE statement. The authorization ID of the statement may be affected by the bind option DYNAMICRULES.

For statements where authorization checking is performed at statement preparation time (DML), no authorization is required to use this statement.

Syntax:



Description:

statement-name
Identifies the prepared statement to be executed. The *statement-name* must identify a statement that was previously prepared, and the prepared statement cannot be a SELECT statement.

INTO
Introduces a list of host variables which are used to receive values from output parameter markers (question marks) in the prepared statement.

For a dynamic CALL statement, parameter markers appearing in OUT and INOUT arguments to the stored procedure are output parameter markers. If any output parameter markers appear in the statement, the

INTO clause must be specified (SQLSTATE 07007). No other dynamic statement has output parameter markers, so this clause may only be used with a result descriptor containing no entries for statements other than CALL (SQLSTATE 07002).

result-host-variable, ...
Identifies a host variable that is declared in the program in accordance with the rules for declaring host variables. The number of variables must be the same as the number of output parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement. Locator variables and file reference variables, where appropriate, can be provided as the destination for parameter markers.

DESCRIPTOR *result-descriptor-name*
Identifies an output SQLDA that must contain a valid description of host variables.

Before the EXECUTE statement is processed, the user must set the following fields in the input SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences.

If LOB or structured data type output data must be accommodated, there must be two SQLVAR entries for every parameter marker.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

USING
Introduces a list of host variables for which values are substituted for the input parameter markers (question marks) in the prepared statement.

For a dynamic CALL statement, parameter markers appearing in IN and INOUT arguments to the stored procedure are input parameter markers. For all other dynamic statements, all the parameter markers are input parameter markers. If any output parameter markers appear in the statement, the USING clause must be specified (SQLSTATE 07004).

input-host-variable, ...
Identifies a host variable that is declared in the program in accordance with the rules for declaring host variables. The number of variables must be the same as the number of input parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement. Locator variables and file reference variables, where appropriate, can be provided as the source of values for parameter markers.

DESCRIPTOR *input-descriptor-name*
Identifies an input SQLDA that must contain a valid description of host variables.

Before the EXECUTE statement is processed, the user must set the following fields in the input SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables.

The 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 LOB or structured data type input data must be accommodated, there must be two SQLVAR entries for every parameter marker.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

Notes:

- Before the prepared statement is executed, each input parameter marker is effectively replaced by the value of its corresponding host variable. 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.

  Let V denote an input host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:

  – V must be compatible with the target.

- If V is a string, its length must not be greater than the length attribute of the target.
- If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.

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.

- For a dynamic CALL statement, after the prepared statement is executed, the returned value of each OUT and INOUT argument is assigned to the host variable corresponding to the output parameter marker used for the argument. 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 those specified by the definition of the parameter of the stored procedure.

  Let V denote an output host variable that corresponds to parameter marker P, which is used for argument A of a stored procedure. The value of A is assigned to V in accordance with the rules for retrieving a value from a column. Thus:

  - V must be compatible with A.
  - If V is a string, its length must not be less than the length of A, or the value of A will be truncated.
  - If V is a number, the maximum absolute value of its integral part must not be less than the absolute value of the integral part of A.
  - If the attributes of V are not identical to the attributes of A, the value of A is converted to conform to the attributes of V.
- ***Dynamic SQL Statement Caching:*** The information required to execute dynamic and static SQL statements is placed in the database package cache when static SQL statements are first referenced or when dynamic SQL statements are first prepared. This information stays in the package cache until it becomes invalid, the cache space is required for another statement, or the database is shut down.

  When an SQL statement is executed or prepared, the package information relevant to the application issuing the request is loaded from the system catalog into the package cache. The actual executable section for the individual SQL statement is also placed into the cache: static SQL sections are read in from the system catalog and placed in the package cache when the statement is first referenced; dynamic SQL sections are placed directly in the cache after they have been created. Dynamic SQL sections can be

created by an explicit statement, such as PREPARE or EXECUTE IMMEDIATE. Once created, sections for dynamic SQL statements may be recreated by an implicit prepare of the statement by the system if the original section has been deleted for space management reasons, or has become invalid due to changes in the environment.

Each SQL statement is cached at the database level and can be shared among applications. Static SQL statements are shared among applications using the same package; dynamic SQL statements are shared among applications using the same compilation environment, and the exact same statement text. The text of each SQL statement issued by an application is cached locally within the application for use if an implicit prepare is required. Each PREPARE statement in the application program can cache one statement. All EXECUTE IMMEDIATE statements in an application program share the same space, and only one cached statement exists for all these EXECUTE IMMEDIATE statements at a time. If the same PREPARE or any EXECUTE IMMEDIATE statement is issued multiple times with a different SQL statement each time, only the last statement will be cached for reuse. The optimal use of the cache is to issue a number of different PREPARE statements once at the start of the application, and then to issue an EXECUTE or OPEN statement as required.

With the caching of dynamic SQL statements, once a statement has been created, it can be reused over multiple units of work without the need to prepare the statement again. The system will recompile the statement, as required, if environment changes occur.

The following events are examples of environment or data object changes that can cause cached dynamic statements to be implicitly prepared on the next PREPARE, EXECUTE, EXECUTE IMMEDIATE, or OPEN request:

- ALTER FUNCTION
- ALTER METHOD
- ALTER NICKNAME
- ALTER PROCEDURE
- ALTER SERVER
- ALTER TABLE
- ALTER TABLESPACE
- ALTER TYPE
- CREATE FUNCTION
- CREATE FUNCTION MAPPING
- CREATE INDEX
- CREATE METHOD
- CREATE PROCEDURE
- CREATE TABLE

- CREATE TEMPORARY TABLESPACE
- CREATE TRIGGER
- CREATE TYPE
- DROP (all objects)
- RUNSTATS on any table or index
- Any action that causes a view to become inoperative
- UPDATE of statistics in any system catalog table
- SET CURRENT DEGREE
- SET PATH
- SET QUERY OPTIMIZATION
- SET SCHEMA
- SET SERVER OPTION

The following list outlines the behavior that can be expected from cached dynamic SQL statements:

- *PREPARE Requests:* Subsequent preparations of the same statement will not incur the cost of compiling the statement if the section is still valid. The cost and cardinality estimates for the current cached section will be returned. These values may differ from the values returned from any previous PREPARE for the same SQL statement. There will be no need to issue a PREPARE statement subsequent to a COMMIT or ROLLBACK statement.
- *EXECUTE Requests:* EXECUTE statements may occasionally incur the cost of implicitly preparing the statement if it has become invalid since the original PREPARE. If a section is implicitly prepared, it will use the current environment and not the environment of the original PREPARE statement.
- *EXECUTE IMMEDIATE Requests:* Subsequent EXECUTE IMMEDIATE statements for the same statement will not incur the cost of compiling the statement if the section is still valid.
- *OPEN Requests:* OPEN requests for dynamically defined cursors may occasionally incur the cost of implicitly preparing the statement if it has become invalid since the original PREPARE statement. If a section is implicitly prepared, it will use the current environment and not the environment of the original PREPARE statement.
- *FETCH Requests:* No behavior changes should be expected.
- *ROLLBACK:* Only those dynamic SQL statements prepared or implicitly prepared during the unit of work affected by the rollback operation will be invalidated.
- *COMMIT:* Dynamic SQL statements will not be invalidated, but any acquired locks will be freed. Cursors not defined as WITH HOLD

cursors will be closed and their locks freed. Open WITH HOLD cursors will hold onto their package and section locks to protect the active section both during and after commit processing.

If an error occurs during an implicit prepare, an error will be returned for the request causing the implicit prepare (SQLSTATE 56098).

Examples:

Example 1: In this C example, an INSERT statement with parameter markers is prepared and executed. Host variables `h1 - h4` correspond to the format of TDEPT.

```
strcpy (s,"INSERT INTO TDEPT VALUES(?,?,?,?)");
EXEC SQL PREPARE DEPT_INSERT FROM :s;
  .
  .
(Check for successful execution and put values into :h1, :h2, :h3, :h4)
  .
  .
EXEC SQL EXECUTE DEPT_INSERT USING :h1, :h2,
:h3, :h4;
```

Example 2: This EXECUTE statement uses an SQLDA.

```
EXECUTE S3 USING DESCRIPTOR :sqlda3
```

Example 3: Given a stored procedure to award an employee a bonus:

```
CREATE PROCEDURE GIVE_BONUS (IN EMPNO INTEGER,
                             IN DEPTNO INTEGER,
                             OUT CHEQUE INTEGER,
                             INOUT BONUS DEC(6,0))
...
```

Dynamically call the stored procedure from a C application. The stored procedure takes the following host variables as input:

- *employee*, the ID number of the employee
- *dept*, the department number
- *bonus*, the desired bonus for the employee

The stored procedure returns the following values to the host variables:

- *cheque_no*, the ID number from the cheque
- *bonus*, the actual bonus amount (after any adjustments)

```
strcpy (s, "CALL GIVE_BONUS(?, ?, ?, ?)");
EXEC SQL PREPARE DO_BONUS FROM :s;
  .
  .
/* Check for successful execution and put values into
   :employee, :dept, and :bonus */
  .
```

```
  .
EXEC SQL EXECUTE DO_BONUS INTO :cheque_no, :bonus
                          USING :employee, :dept, :bonus;
  .
  .
/* Check for successful execution and process the
   values returned in :cheque_no and :bonus */
```

Related reference:

- "Identifiers" in the *SQL Reference, Volume 1*
- "PREPARE" on page 620
- "SQLDA (SQL descriptor area)" in the *SQL Reference, Volume 1*

Related samples:

- "dbuse.sqc -- How to use a database (C)"
- "fnuse.sqc -- How to use built-in SQL functions (C)"
- "tut_use.sqc -- How to modify a database (C)"
- "udfcli.sqc -- Call a variety of types of user-defined functions (C)"
- "dbuse.sqC -- How to use a database (C++)"
- "fnuse.sqC -- How to use built-in SQL functions (C++)"
- "tut_use.sqC -- How to modify a database (C++)"
- "udfcli.sqC -- Call a variety of types of user-defined functions (C++)"
- "inpsrv.sqb -- Demonstrates stored procedures using the SQLDA structure (MF COBOL)"

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 statements that contain neither host 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.

Authorization:

The authorization rules are those defined for the specified SQL statement.

The authorization ID of the statement may be affected by the DYNAMICRULES bind option.

Syntax:

Description:

host-variable

A host variable must be specified, and it must identify a host variable that is described in the program in accordance with the rules for declaring character-string variables. It must be a character-string variable that is less than the maximum statement size of 65 535. Note that a CLOB(65535) can contain a maximum size statement, but a VARCHAR can not. The value of the identified host variable is called the statement string.

The statement string must be one of the following SQL statements:

- ALTER
- CALL
- COMMENT
- COMMIT
- CREATE
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE

- DROP
- GRANT
- INSERT
- LOCK TABLE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME TABLE
- RENAME TABLESPACE
- REVOKE
- ROLLBACK
- SAVEPOINT
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET SCHEMA
- SET SERVER OPTION
- UPDATE

The statement string must not include parameter markers or references to host variables, and must not begin with EXEC SQL. It must not contain a statement terminator, with the exception of the CREATE TRIGGER and CREATE PROCEDURE statements. A CREATE TRIGGER statement can contain semi-colons (;) to separate triggered SQL statements. A CREATE PROCEDURE statement can contain semi-colons to separate SQL statements in the SQL procedure body. The stored procedure named in a CALL statement must not have any OUT or INOUT parameters (SQLSTATE 07007).

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 condition that prevents its

execution is reported in the SQLCA. If the SQL statement is valid, but an error occurs during its execution, that error condition is reported in the SQLCA.

Notes:

- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement.

Example:

Use C program statements to move an SQL statement to the host variable qstring (char[80]), and prepare and execute whatever SQL statement is in the host variable qstring.

```
if ( strcmp(accounts,"BIG") == 0 )
  strcpy (qstring,"INSERT INTO WORK_TABLE SELECT *
    FROM EMP_ACT WHERE ACTNO < 100");
else
  strcpy (qstring,"INSERT INTO WORK_TABLE SELECT *
    FROM EMP_ACT WHERE ACTNO >= 100");
  .
  .
  .
EXEC SQL  EXECUTE IMMEDIATE :qstring;
```

Related reference:

- "Identifiers" in the *SQL Reference, Volume 1*
- "EXECUTE" on page 544

Related samples:

- "dbuse.sqc -- How to use a database (C)"
- "dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)"
- "fnuse.sqc -- How to use built-in SQL functions (C)"
- "tbconstr.sqc -- How to create, use, and drop constraints (C)"
- "tbtrig.sqc -- How to use a trigger on a table (C)"
- "tscreate.sqc -- How to create and drop buffer pools and table spaces (C)"
- "dbuse.sqC -- How to use a database (C++)"
- "dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)"
- "dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)"
- "fnuse.sqC -- How to use built-in SQL functions (C++)"
- "tbconstr.sqC -- How to create, use, and drop constraints (C++)"
- "tbtrig.sqC -- How to use a trigger on a table (C++)"

- "tscreate.sqC -- How to create and drop buffer pools and table spaces (C++)"
- "DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)"
- "expsamp.sqb -- Export and import tables with table data to a DRDA database (MF COBOL)"

EXPLAIN

The EXPLAIN statement captures information about the access plan chosen for the supplied explainable statement and places this information into the Explain tables.

An *explainable statement* is a DELETE, INSERT, SELECT, SELECT INTO, UPDATE, VALUES, or VALUES INTO SQL statement.

Invocation:

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

The statement to be explained is not executed.

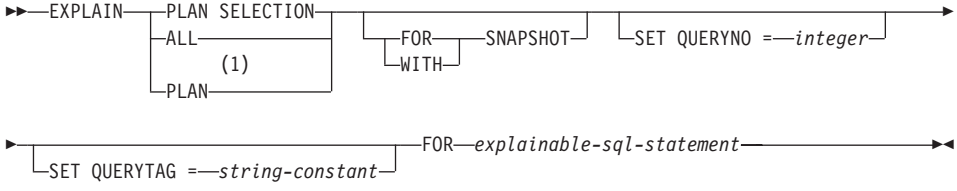
Authorization:

The authorization rules are those defined for the SQL statement specified in the EXPLAIN statement. For example, if a DELETE statement was used as the *explainable-sql-statement* (see statement syntax that follows), then the authorization rules for a DELETE statement would be applied when the DELETE statement is explained.

The authorization rules for static EXPLAIN statements are those rules that apply for static versions of the statement passed as the *explainable-sql-statement*. Dynamically prepared EXPLAIN statements use the authorization rules for the dynamic preparation of the statement provided for the *explainable-sql-statement* parameter.

The current authorization ID must have insert privilege on the Explain tables.

Syntax:



Notes:

1 The PLAN option is supported only for syntax toleration of existing DB2 for MVS EXPLAIN statements. There is no PLAN table.
Specifying PLAN is equivalent to specifying PLAN SELECTION.

Description:

PLAN SELECTION
Indicates that the information from the plan selection phase of SQL compilation is to be inserted into the Explain tables.

ALL
Specifying ALL is equivalent to specifying PLAN SELECTION.

PLAN
The PLAN option provides syntax toleration for existing database applications from other systems. Specifying PLAN is equivalent to specifying PLAN SELECTION.

FOR SNAPSHOT
This clause indicates that only an Explain Snapshot is to be taken and placed into the SNAPSHOT column of the EXPLAIN_STATEMENT table. No other Explain information is captured other than that present in the EXPLAIN_INSTANCE and EXPLAIN_STATEMENT tables.

The Explain Snapshot information is intended for use with Visual Explain.

WITH SNAPSHOT
This clause indicates that, in addition to the regular Explain information, an Explain Snapshot is to be taken.

The default behavior of the EXPLAIN statement is to only gather regular Explain information and not the Explain Snapshot.

The Explain Snapshot information is intended for use with Visual Explain.

default (neither FOR SNAPSHOT nor WITH SNAPSHOT specified)
Puts Explain information into the Explain tables. No snapshot is taken for use with Visual Explain.

SET QUERYNO = *integer*
Associates *integer*, via the QUERYNO column in the EXPLAIN_STATEMENT table, with *explainable-sql-statement*. The integer value supplied must be a positive value.

If this clause is not specified for a dynamic EXPLAIN statement, a default value of one (1) is assigned. For a static EXPLAIN statement, the default value assigned is the statement number assigned by the precompiler.

SET QUERYTAG = *string-constant*
Associates *string-constant*, via the QUERYTAG column in the EXPLAIN_STATEMENT table, with *explainable-sql-statement*. *string-constant* can be any character string up to 20 bytes in length. If the value supplied is less than 20 bytes in length, the value is padded on the right with blanks to the required length.

If this clause is not specified for an EXPLAIN statement, blanks are used as the default value.

FOR *explainable-sql-statement*
Specifies the SQL statement to be explained. This statement can be any valid DELETE, INSERT, SELECT, SELECT INTO, UPDATE, VALUES, or VALUES INTO SQL statement. If the EXPLAIN statement is embedded in a program, the *explainable-sql-statement* can contain references to host variables (these variables must be defined in the program). Similarly, if EXPLAIN is being dynamically prepared, the *explainable-sql-statement* can contain parameter markers.

The *explainable-sql-statement* must be a valid SQL statement that could be prepared and executed independently of the EXPLAIN statement. It cannot be a statement name or host variable. SQL statements referring to cursors defined through CLP are not valid for use with this statement.

To explain dynamic SQL within an application, the entire EXPLAIN statement must be dynamically prepared.

Notes:

The following table shows the interaction of the snapshot keywords and the Explain information.

Keyword Specified	Capture Explain Information?	Take Snapshot for Visual Explain?
none	Yes	No
FOR SNAPSHOT	No	Yes
WITH SNAPSHOT	Yes	Yes

If neither the FOR SNAPSHOT nor the WITH SNAPSHOT clause is specified, an Explain snapshot is not taken.

The Explain tables must be created by the user prior to invocation of the EXPLAIN statement. The information generated by this statement is stored in the Explain tables, in the schema that is designated at the time the statement is compiled.

If any errors occur during the compilation of the *explainable-sql-statement* supplied, then no information is stored in the Explain tables.

The access plan generated for the *explainable-sql-statement* is not saved and thus, cannot be invoked at a later time. The Explain information for the *explainable-sql-statement* is inserted when the EXPLAIN statement itself is compiled.

For a static EXPLAIN SQL statement, the information is inserted into the Explain tables at bind time and during an explicit rebind. During precompilation, the static EXPLAIN statements are commented out in the modified application source file. At bind time, the EXPLAIN statements are stored in the SYSCAT.STATEMENTS catalog. When the package is run, the EXPLAIN statement is not executed. Note that the section numbers for all statements in the application will be sequential and will include the EXPLAIN statements. An alternative to using a static EXPLAIN statement is to use a combination of the EXPLAIN and EXPLSNAP BIND/PREP options. Static EXPLAIN statements can be used to cause the Explain tables to be populated for one specific static SQL statement out of many; simply prefix the target statement with the appropriate EXPLAIN statement syntax and bind the application without using either of the Explain BIND/PREP options. The EXPLAIN statement can also be used when it is advantageous to set the QUERYNO or QUERYTAG field at the time of the actual Explain invocation.

For an incremental bind EXPLAIN SQL statement, the Explain tables are populated when the EXPLAIN statement is submitted for compilation. When the package is run, the EXPLAIN statement performs no processing (though the statement will be successful). When populating the explain tables, the explain table qualifier and authorization ID used during population will be those of the package owner. The EXPLAIN statement can also be used when it is advantageous to set the QUERYNO or QUERYTAG field at the time of the actual Explain invocation.

For dynamic EXPLAIN statements, the Explain tables are populated at the time the EXPLAIN statement is submitted for compilation. An Explain statement can be prepared with the PREPARE statement but, if executed, will perform no processing (though the statement will be successful). An alternative to issuing dynamic EXPLAIN statements is to use a combination of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special registers to explain dynamic SQL statements. The EXPLAIN statement should be used when it is advantageous to set the QUERYNO or QUERYTAG field at the time of the actual Explain invocation.

Examples:

Example 1: Explain a simple SELECT statement and tag with QUERYNO = 13.

```
EXPLAIN PLAN SET QUERYNO = 13 FOR SELECT C1 FROM T1;
```

This statement is successful.

Example 2:

Explain a simple SELECT statement and tag with QUERYTAG = 'TEST13'.

```
EXPLAIN PLAN SELECTION SET QUERYTAG = 'TEST13'
  FOR SELECT C1 FROM T1;
```

This statement is successful.

Example 3: Explain a simple SELECT statement and tag with QUERYNO = 13 and QUERYTAG = 'TEST13'.

```
EXPLAIN PLAN SELECTION SET QUERYNO = 13 SET QUERYTAG = 'TEST13'
  FOR SELECT C1 FROM T1;
```

This statement is successful.

Example 4: Attempt to get Explain information when Explain tables do not exist.

```
EXPLAIN ALL FOR SELECT C1 FROM T1;
```

This statement would fail as the Explain tables have not been defined (SQLSTATE 42704).

Related reference:

- "EXPLAIN_ARGUMENT table" in the *SQL Reference, Volume 1*
- "EXPLAIN_OBJECT table" in the *SQL Reference, Volume 1*
- "EXPLAIN_OPERATOR table" in the *SQL Reference, Volume 1*
- "EXPLAIN_PREDICATE table" in the *SQL Reference, Volume 1*
- "EXPLAIN_STREAM table" in the *SQL Reference, Volume 1*
- "ADVISE_INDEX table" in the *SQL Reference, Volume 1*
- "ADVISE_WORKLOAD table" in the *SQL Reference, Volume 1*
- "EXPLAIN_INSTANCE table" in the *SQL Reference, Volume 1*
- "EXPLAIN_STATEMENT table" in the *SQL Reference, Volume 1*
- "Explain tables" in the *SQL Reference, Volume 1*

FETCH

The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to host variables.

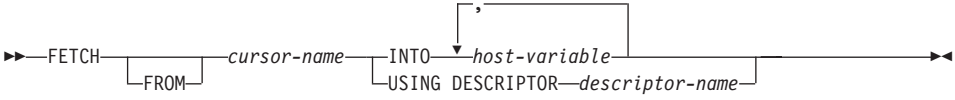
Invocation:

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:

For an explanation of the authorization required to use a cursor, see "DECLARE CURSOR".

Syntax:



Description:

cursor-name
Identifies the cursor to be used in the fetch operation. The *cursor-name* must identify a declared cursor, as explained in "DECLARE CURSOR". The DECLARE CURSOR statement must precede the FETCH statement in the source program. When the FETCH statement is executed, the cursor must be in the open state.

If the cursor is currently positioned on or after the last row of the result table:

- SQLCODE is set to +100, and SQLSTATE is set to '02000'.
- The cursor is positioned after the last row.
- Values are not assigned to host variables.

If the cursor is currently positioned before a row, it will be repositioned on that row, and values will be assigned to host variables as specified by INTO or USING.

If the cursor is currently positioned on a row other than the last row, it will be repositioned on the next row and values of that row will be assigned to host variables as specified by INTO or USING.

INTO *host-variable*, **...**
Identifies one or more host variables that must be described in accordance with the rules for declaring host variables. The first value in the result row is assigned to the first host variable in the list, the second value to the second host variable, and so on. For LOB values in the select-list, the target can be a regular host variable (if it is large enough), a locator variable, or a file-reference variable.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more host variables.

Before the FETCH statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of the variables.

The 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 LOB or structured type result columns need to be accommodated, there must be two SQLVAR entries for every select-list item (or column of the result table).

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

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 specific rules. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLDA is set to 'W'. Note that there is no warning if there are more variables than the number of result columns. 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.

Notes:

- An open cursor has three possible positions:
  - Before a row
  - On a row
  - After the last row.
- 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.
- When retrieving into LOB locators in situations where it is not necessary to retain the locator across FETCH statements, it is good practice to issue a FREE LOCATOR statement before issuing the next FETCH statement, as locator resources are limited.
- It is possible for an error to occur that makes the state of the cursor unpredictable.
- It is possible that a warning may not be returned on a FETCH. It is also possible that the returned warning applies to a previously fetched row. This occurs as a result of optimizations such as the use of system temporary tables or pushdown operators.
- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement.
- DB2 CLI supports additional fetching capabilities. For instance when a cursor's result table is read-only, the SQLFetchScroll() function can be used to position the cursor at any spot within that result table.
- For an updatable cursor, a lock is obtained on a row when it is fetched.

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
```

Related reference:

- "DECLARE CURSOR" on page 482
- "EXECUTE" on page 544
- "SQLDA (SQL descriptor area)" in the *SQL Reference, Volume 1*
- "Assignments and comparisons" in the *SQL Reference, Volume 1*

Related samples:

- "cursor.sqb -- How to update table data with cursor statically (MF COBOL)"
- "tbread.sqc -- How to read tables (C)"
- "tut_mod.sqc -- How to modify table data (C)"
- "tbread.sqC -- How to read tables (C++)"
- "tut_mod.sqC -- How to modify table data (C++)"
- "TutMod.sqlj -- Modify data in a table (SQLj)"

FLUSH EVENT MONITOR

The FLUSH EVENT MONITOR statement writes current database monitor values for all active monitor types associated with event monitor *event-monitor-name* to the event monitor I/O target. Hence, at any time a partial event record is available for event monitors that have low record generation frequency (such as a database event monitor). Such records are noted in the event monitor log with a *partial record* identifier.

When an event monitor is flushed, its active internal buffers are written to the event monitor output object.

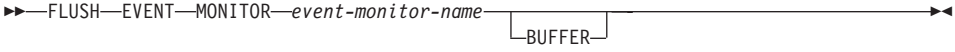
Invocation:

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization:

The privileges held by the authorization ID must include either SYSADM or DBADM authority (SQLSTATE 42502).

Syntax:



Description:

event-monitor-name
Name of the event monitor. This is a one-part name. It is an SQL identifier.

BUFFER
Indicates that the event monitor buffers are to be written out. If BUFFER is specified, then a partial record is not generated. Only the data already present in the event monitor buffers are written out.

Notes:

- Flushing out the event monitor will not cause the event monitor values to be reset. This means that the event monitor record that would have been generated if no flush was performed, will still be generated when the normal monitor event is triggered.

FLUSH PACKAGE CACHE

The FLUSH PACKAGE CACHE statement removes all cached dynamic SQL statements currently in the package cache. This statement causes the logical invalidation of any cached dynamic SQL statement and forces the next request for the same SQL statement to be implicitly compiled by DB2.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

The privileges held by the authorization ID of the statement must include either SYSADM or DBADM authority (SQLSTATE 42502).

Syntax:

Notes:

- This statement affects all cached dynamic SQL entries in the package cache on all active database partitions.
- As cached dynamic SQL statements are invalidated, the package cache memory used for the cached entry will be freed if the entry is not in use when the FLUSH PACKAGE CACHE statement executes.
- Any cached dynamic SQL statement currently in use will be allowed to continue to exist in the package cache until it is no longer needed by the its current user; the next new user of the same statement will force an implicit prepare of the statement by DB2, and the new user will execute the new version of the cached dynamic SQL statement.

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 cannot be dynamically prepared.

Authorization:

None required.

Syntax:

Description:

LOCATOR *variable-name*, **...**
Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables.

The locator-variable must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a CALL, FETCH, SELECT INTO, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is returned (SQLSTATE 0F001).

If more than one locator is specified, all locators that can be freed will be freed, regardless of errors detected in other locators in the list.

Example:

In a COBOL program, free the BLOB locator variables TKN-VIDEO and TKN-BUF and the CLOB locator variable LIFE-STORY-LOCATOR.

```
EXEC SQL
FREE LOCATOR :TKN-VIDEO, :TKN-BUF, :LIFE-STORY-LOCATOR
END-EXEC.
```

Related samples:

- "dtlob.c -- How to read and write LOB data (CLI)"
- "spserver.c -- Definition of various types of stored procedures (CLI)"
- "dtlob.sqc -- How to use the LOB data type (C)"

- “spserver.sqc -- A variety of types of stored procedures (C)”
- “dtlob.sqC -- How to use the LOB data type (C++)”
- “spserver.sqC -- A variety of types of stored procedures (C++)”
- “lobeval.sqb -- Demonstrates how to use a Large Object (LOB) (MF COBOL)”

GRANT (Database Authorities)

This form of the GRANT statement grants authorities that apply to the entire database (rather than privileges that apply to specific objects within the database).

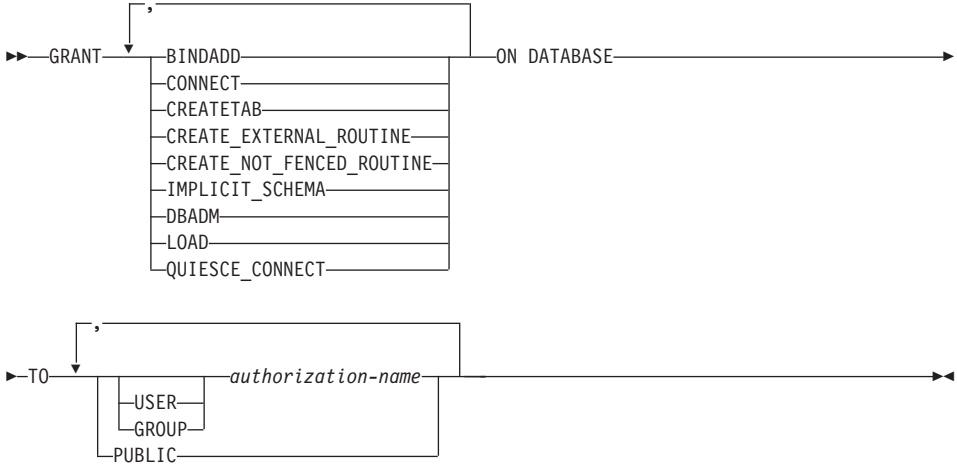
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

To grant DBADM authority, SYSADM authority is required. To grant other authorities, either DBADM or SYSADM authority is required.

Syntax:



Description:

BINDADD
Grants the authority to create packages. The creator of a package automatically has the CONTROL privilege on that package and retains this privilege even if the BINDADD authority is subsequently revoked.

CONNECT
Grants the authority to access the database.

CREATETAB
Grants the authority to create base tables. The creator of a base table automatically has the CONTROL privilege on that table. The creator retains this privilege even if the CREATETAB authority is subsequently revoked.

There is no explicit authority required for view creation. A view can be created at any time if the authorization ID of the statement used to create the view has either CONTROL or SELECT privilege on each base table of the view.

CREATE_EXTERNAL_ROUTINE
Grants the authority to register external routines. Care must be taken that routines so registered will not have adverse side effects. (For more information, see the description of the THREADSAFE clause on the CREATE or ALTER routine statements.)

Once an external routine has been registered, it continues to exist, even if CREATE_EXTERNAL_ROUTINE is subsequently revoked.

CREATE_NOT_FENCED_ROUTINE
Grants the authority to register routines that execute in the database manager's process. Care must be taken that routines so registered will not have adverse side effects. (For more information, see the description of the FENCED clause on the CREATE or ALTER routine statements.)

Once a routine has been registered as not fenced, it continues to run in this manner, even if CREATE_NOT_FENCED_ROUTINE is subsequently revoked.

CREATE_EXTERNAL_ROUTINE is automatically granted to an *authorization-name* that is granted CREATE_NOT_FENCED_ROUTINE authority.

IMPLICIT_SCHEMA
Grants the authority to implicitly create a schema.

DBADM
Grants the database administrator authority and all other database authorities. A database administrator has all privileges against all objects in the database, and can grant these privileges to others.

Note: All other database authorities are implicitly and automatically granted to an *authorization-name* that is granted DBADM authority.

LOAD
Grants the authority to load in this database. This authority gives a user the right to use the LOAD utility in this database. SYSADM and DBADM also have this authority by default. However, if a user only has LOAD

authority (not SYSADM or DBADM), the user is also required to have table-level privileges. In addition to LOAD privilege, the user is required to have:

- INSERT privilege on the table for LOAD with mode INSERT, TERMINATE (to terminate a previous LOAD INSERT), or RESTART (to restart a previous LOAD INSERT)
- INSERT and DELETE privilege on the table for LOAD with mode REPLACE, TERMINATE (to terminate a previous LOAD REPLACE), or RESTART (to restart a previous LOAD REPLACE)
- INSERT privilege on the exception table, if such a table is used as part of LOAD

QUIESCE_CONNECT
Grants the authority to access the database while it is quiesced.

TO
Specifies to whom the authorities are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the authorities to all users. DBADM cannot be granted to PUBLIC.

Rules:

- If neither USER nor GROUP is specified, then
  - If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:

- CREATE_NOT_FENCED can be specified in place of CREATE_NOT_FENCED_ROUTINE

Examples:

Example 1: Give the users WINKEN, BLINKEN, and NOD the authority to connect to the database.

```
GRANT CONNECT ON DATABASE TO USER WINKEN, USER BLINKEN, USER NOD
```

Example 2: GRANT BINDADD authority on the database to a group named D024. There is both a group and a user called D024 in the system.

```
GRANT BINDADD ON DATABASE TO GROUP D024
```

Observe that, the GROUP keyword must be specified; otherwise, an error will occur since both a user and a group named D024 exist. Any member of the D024 group will be allowed to bind packages in the database, but the D024 user will not be allowed (unless this user is also a member of the group D024, had been granted BINDADD authority previously, or BINDADD authority had been granted to another group of which D024 was a member).

Related reference:

- "GRANT (Index Privileges)" on page 573
- "GRANT (Package Privileges)" on page 575
- "GRANT (Schema Privileges)" on page 583
- "GRANT (Table, View, or Nickname Privileges)" on page 590
- "GRANT (Server Privileges)" on page 588
- "GRANT (Table Space Privileges)" on page 598
- "GRANT (Sequence Privileges)" on page 586
- "GRANT (Routine Privileges)" on page 579

Related samples:

- "dbauth.sqb -- How to grant and display authorities on a database (MF COBOL)"
- "dbauth.sqc -- How to grant, display, and revoke authorities at database level (C)"
- "dbauth.sqC -- How to grant, display, and revoke authorities at database level (C++)"
- "DbAuth.java -- Grant, display or revoke privileges on database (JDBC)"
- "DbAuth.sqlj -- Grant, display or revoke privileges on database (SQLj)"

GRANT (Index Privileges)

This form of the GRANT statement grants the CONTROL privilege on indexes.

Invocation:

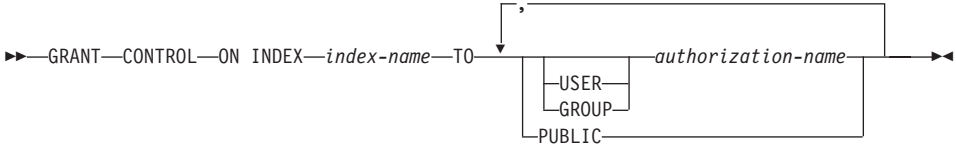
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- DBADM authority
- SYSADM authority.

Syntax:



Description:

CONTROL
Grants the privilege to drop the index. This is the CONTROL authority for indexes, which is automatically granted to creators of indexes.

ON INDEX *index-name*
Identifies the index for which the CONTROL privilege is to be granted.

TO
Specifies to whom the privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the privileges to all users.

Rules:

- If neither USER nor GROUP is specified, then
  - If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

Example:

```
GRANT CONTROL ON INDEX DEPTIDX TO USER USER4
```

Related reference:

- "GRANT (Database Authorities)" on page 569
- "GRANT (Package Privileges)" on page 575
- "GRANT (Schema Privileges)" on page 583
- "GRANT (Table, View, or Nickname Privileges)" on page 590
- "GRANT (Server Privileges)" on page 588
- "GRANT (Table Space Privileges)" on page 598
- "GRANT (Sequence Privileges)" on page 586
- "GRANT (Routine Privileges)" on page 579

GRANT (Package Privileges)

This form of the GRANT statement grants privileges on a package.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

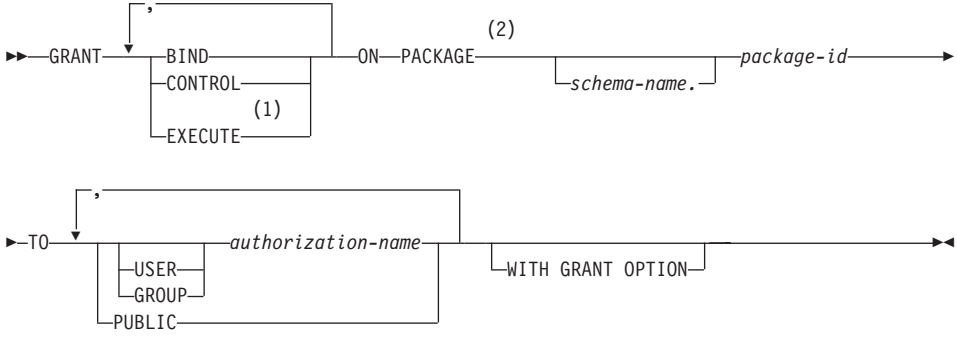
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- CONTROL privilege on the referenced package
- The WITH GRANT OPTION for each identified privilege on *package-name*.
- SYSADM or DBADM authority.

To grant the CONTROL privilege, SYSADM or DBADM authority is required.

Syntax:



Notes:

1 RUN can be used as a synonym for EXECUTE.

2 PROGRAM can be used as a synonym for PACKAGE.

Description:

BIND
Grants the privilege to bind a package. The BIND privilege allows a user

to re-issue the BIND command against that package, or to issue the REBIND command. It also allows a user to create a new version of an existing package.

In addition to the BIND privilege, a user must hold the necessary privileges on each table referenced by static DML statements contained in a program. This is necessary, because authorization on static DML statements is checked at bind time.

CONTROL
Grants the privilege to rebind, drop, or execute the package, and extend package privileges to other users. The CONTROL privilege for packages is automatically granted to creators of packages. A package owner is the package binder, or the ID specified with the OWNER option at bind/precompile time.

BIND and EXECUTE are automatically granted to an *authorization-name* that is granted CONTROL privilege.

CONTROL grants the ability to grant the above privileges (except for CONTROL) to others.

EXECUTE
Grants the privilege to execute the package.

ON PACKAGE *schema-name.package-id*
Specifies the name of the package on which privileges are to be granted. If a schema name is not specified, the package ID is implicitly qualified by the default schema. The granting of a package privilege applies to all versions of the package (that is, to all packages that share the same package ID and package schema).

TO
Specifies to whom the privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the privileges to all users.

WITH GRANT OPTION
Allows the specified *authorization-name* to GRANT the privileges to others.

If the specified privileges include CONTROL, the WITH GRANT OPTION applies to all of the applicable privileges except for CONTROL (SQLSTATE 01516).

Rules:

- If neither USER nor GROUP is specified, then
  - If the *authorization-name* is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the *authorization-name* is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is returned.

Notes:

- Package privileges apply to all versions of a package (that is, all packages that share the same package ID and package schema). It is not possible to restrict access to only one version. Because CONTROL privilege is implicitly granted to the binder of a package, if two different users bind two versions of a package, then both users will implicitly be granted access to each other's package.

Examples:

Example 1: Grant the EXECUTE privilege on PACKAGE CORPDATA.PKGA to PUBLIC.

```
GRANT EXECUTE
  ON PACKAGE CORPDATA.PKGA
  TO PUBLIC
```

Example 2: GRANT EXECUTE privilege on package CORPDATA.PKGA to a user named EMPLOYEE. There is neither a group nor a user called EMPLOYEE.

```
GRANT EXECUTE ON PACKAGE
  CORPDATA.PKGA TO EMPLOYEE
```

or

```
GRANT EXECUTE ON PACKAGE
  CORPDATA.PKGA TO USER EMPLOYEE
```

Related reference:

- "GRANT (Database Authorities)" on page 569

GRANT (Package Privileges)

- "GRANT (Index Privileges)" on page 573
- "GRANT (Schema Privileges)" on page 583
- "GRANT (Table, View, or Nickname Privileges)" on page 590
- "GRANT (Server Privileges)" on page 588
- "GRANT (Table Space Privileges)" on page 598
- "GRANT (Sequence Privileges)" on page 586
- "GRANT (Routine Privileges)" on page 579

GRANT (Routine Privileges)

This form of the GRANT statement grants privileges on a routine (function, method, or procedure).

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

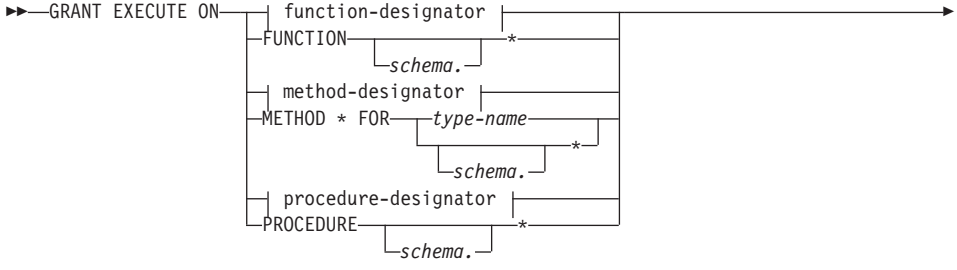
The privileges held by the authorization ID of the statement must include at least one of the following:

- WITH GRANT OPTION for EXECUTE on the routine
- SYSADM or DBADM authority

To grant all routine EXECUTE privileges in the schema or type, the privileges held by the authorization ID must include at least one of the following:

- WITH GRANT OPTION for EXECUTE on all existing and future routines (of the specified type) in the specified schema
- SYSADM or DBADM authority

Syntax:



Description:

EXECUTE
Grants the privilege to run the identified user-defined function, method, or stored procedure.

function-designator
Uniquely identifies the function.

FUNCTION *schema*.*
Identifies all the functions in the schema, including any functions that may be created in the future. In dynamic SQL statements, if a schema is not specified, the schema in the CURRENT SCHEMA special register will be used. In static SQL statements, if a schema is not specified, the schema in the QUALIFIER precompile/bind option will be used.

method-designator
Uniquely identifies the method.

METHOD *
Identifies all the methods for the type *type-name*, including any methods that may be created in the future.

FOR *type-name*
Names the type in which the specified method is found. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the value of the CURRENT SCHEMA special register is used as a qualifier for an unqualified type name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified type names. An asterisk (*) can be used in place of *type-name* to identify all types in the schema, including any types that may be created in the future.

procedure-designator
Uniquely identifies the procedure.

PROCEDURE *schema*.*
Identifies all the procedures in the schema, including any procedures that may be created in the future. In dynamic SQL statements, if a schema is not specified, the schema in the CURRENT SCHEMA special register will be used. In static SQL statements, if a schema is not specified, the schema in the QUALIFIER precompile/bind option will be used.

TO
Specifies to whom the EXECUTE privilege is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

PUBLIC
Grants the EXECUTE privilege to all users.

WITH GRANT OPTION
Allows the specified *authorization-name*s to GRANT the EXECUTE privilege to others.

If the WITH GRANT OPTION is omitted, the specified *authorization-name* can only grant the EXECUTE privilege to others if they:

- have SYSADM or DBADM authority or
- received the ability to grant the EXECUTE privilege from some other source.

Rules:

- It is not possible to grant the EXECUTE privilege on a function or method defined with schema 'SYSIBM' or 'SYSFUN' (SQLSTATE 42832).
- If neither USER nor GROUP is specified, then:
  - If the *authorization-name* is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the *authorization-name* is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined in the operating system as both USER and GROUP, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

Examples:

Example 1: Grant the EXECUTE privilege on function CALC_SALARY to user JONES. Assume that there is only one function in the schema with function name CALC_SALARY.

```
GRANT EXECUTE ON FUNCTION CALC_SALARY TO JONES
```

Example 2: Grant the EXECUTE privilege on procedure VACATION_ACCR to all users at the current server.

```
GRANT EXECUTE ON PROCEDURE VACATION_ACCR TO PUBLIC
```

Example 3: Grant the EXECUTE privilege on function DEPT_TOTALS to the administrative assistant and give the assistant the ability to grant the EXECUTE privilege on this function to others. The function has the specific name DEPT85_TOT. Assume that the schema has more than one function named DEPT_TOTALS.

```
GRANT EXECUTE ON SPECIFIC FUNCTION DEPT85_TOT
  TO ADMIN_A WITH GRANT OPTION
```

Example 4: Grant the EXECUTE privilege on function NEW_DEPT_HIRES to HR (Human Resources). The function has two input parameters of type INTEGER and CHAR(10), respectively. Assume that the schema has more than one function named NEW_DEPT_HIRES.

```
GRANT EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10)) TO HR
```

Example 5: Grant the EXECUTE privilege on method SET_SALARY of type EMPLOYEE to user JONES.

```
GRANT EXECUTE ON METHOD SET_SALARY FOR EMPLOYEE TO JONES
```

Related reference:

- "GRANT (Database Authorities)" on page 569
- "GRANT (Index Privileges)" on page 573
- "GRANT (Package Privileges)" on page 575
- "GRANT (Schema Privileges)" on page 583
- "GRANT (Table, View, or Nickname Privileges)" on page 590
- "GRANT (Server Privileges)" on page 588
- "GRANT (Table Space Privileges)" on page 598
- "GRANT (Sequence Privileges)" on page 586
- "Common syntax elements" on page xi

GRANT (Schema Privileges)

This form of the GRANT statement grants privileges on a schema.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

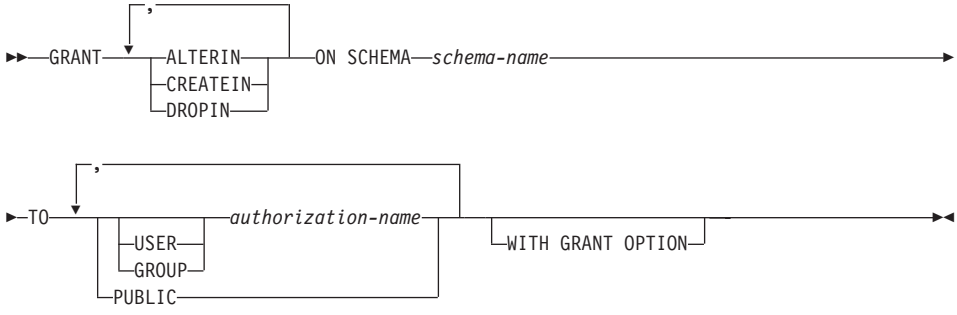
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- WITH GRANT OPTION for each identified privilege on *schema-name*
- SYSADM or DBADM authority

Privileges cannot be granted on schema names SYSIBM, SYSCAT, SYSFUN and SYSSTAT by any user (SQLSTATE 42501).

Syntax:



Description:

ALTERIN
Grants the privilege to alter or comment on all objects in the schema. The owner of an explicitly created schema automatically receives ALTERIN privilege.

CREATEIN
Grants the privilege to create objects in the schema. Other authorities or privileges required to create the object (such as CREATETAB) are still required. The owner of an explicitly created schema automatically receives CREATEIN privilege. An implicitly created schema has CREATEIN privilege automatically granted to PUBLIC.

DROPIN
Grants the privilege to drop all objects in the schema. The owner of an explicitly created schema automatically receives DROPIN privilege.

ON SCHEMA *schema-name*
Identifies the schema on which the privileges are to be granted.

TO
Specifies to whom the privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the privileges to all users.

WITH GRANT OPTION
Allows the specified *authorization-name*s to GRANT the privileges to others.

If the WITH GRANT OPTION is omitted, the specified *authorization-name*s can only grant the privileges to others if they:

- have DBADM authority or
- received the ability to grant privileges from some other source.

Rules:

- If neither USER nor GROUP is specified, then
  - If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.
- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a warning (SQLSTATE 01007) if one or more privileges was not granted. If no privileges were granted, an error is returned (SQLSTATE 42501). (If the package used for processing the statement was precompiled with

LANGLEVEL set to SQL92E for MIA, a warning is returned (SQLSTATE 01007), unless the grantor has no privileges on the object of the grant operation.)

Examples:

Example 1: Grant user JSINGLETON to the ability to create objects in schema CORPDATA.

```
GRANT CREATEIN ON SCHEMA CORPDATA TO JSINGLETON
```

Example 2: Grant user IHAKES the ability to create and drop objects in schema CORPDATA.

```
GRANT CREATEIN, DROPIN ON SCHEMA CORPDATA TO IHAKES
```

Related reference:

- "GRANT (Database Authorities)" on page 569
- "GRANT (Index Privileges)" on page 573
- "GRANT (Package Privileges)" on page 575
- "GRANT (Table, View, or Nickname Privileges)" on page 590
- "GRANT (Server Privileges)" on page 588
- "GRANT (Table Space Privileges)" on page 598
- "GRANT (Sequence Privileges)" on page 586
- "GRANT (Routine Privileges)" on page 579

GRANT (Sequence Privileges)

This form of the GRANT statement grants privileges on a sequence.

Invocation:

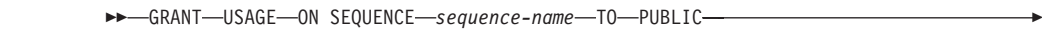
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- WITH GRANT OPTION for use of the table space
- SYSADM or DBADM authority
- Definer of the sequence

Syntax:



Description:

USAGE
Grants the privilege to reference a sequence using *nextval-expression* or *prevval-expression*.

ON SEQUENCE *sequence-name*
Identifies the sequence on which the specified privileges are to be granted. The sequence name, including an implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists, an error (SQLSTATE 42704) is raised.

TO
Specifies to whom the specified privileges are granted.

PUBLIC
Grants the specified privileges to all users.

Example:

Example 1: Grant any user the USAGE privilege on a sequence called ORG_SEQ.

```
GRANT USAGE ON SEQUENCE ORG_SEQ TO PUBLIC
```

Related reference:

- "GRANT (Database Authorities)" on page 569
- "GRANT (Index Privileges)" on page 573
- "GRANT (Package Privileges)" on page 575
- "GRANT (Schema Privileges)" on page 583
- "GRANT (Table, View, or Nickname Privileges)" on page 590
- "GRANT (Server Privileges)" on page 588
- "GRANT (Table Space Privileges)" on page 598
- "GRANT (Routine Privileges)" on page 579

GRANT (Server Privileges)

This form of the GRANT statement grants the privilege to access and use a specified data source in pass-through mode.

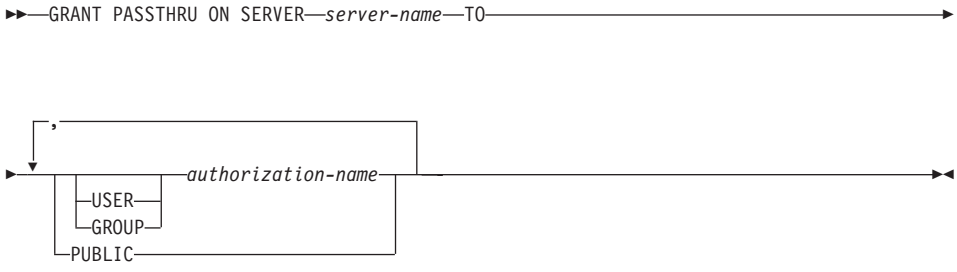
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement must have either SYSADM or DBADM authority.

Syntax:



Description:

server-name
Names the data source for which the privilege to use in pass-through mode is being granted. *server-name* must identify a data source that is described in the catalog.

TO
Specifies to whom the privilege is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants to all users the privilege to pass through to *server-name.*

Examples:

Example 1: Give R. Smith and J. Jones the privilege to pass through to data source SERVALL. Their authorization IDs are RSMITH and JJONES.

```
   GRANT PASSTHRU ON SERVER SERVALL
TO USER RSMITH,
USER JJONES
```

Example 2: Grant the privilege to pass through to data source EASTWING to a group whose authorization ID is D024. There is a user whose authorization ID is also D024.

```
   GRANT PASSTHRU ON SERVER EASTWING TO GROUP D024
```

The GROUP keyword must be specified; otherwise, an error will occur because D024 is a user's ID as well as the specified group's ID (SQLSTATE 56092). Any member of group D024 will be allowed to pass through to EASTWING. Therefore, if user D024 belongs to the group, this user will be able to pass through to EASTWING.

Related reference:

- "GRANT (Database Authorities)" on page 569
- "GRANT (Index Privileges)" on page 573
- "GRANT (Package Privileges)" on page 575
- "GRANT (Schema Privileges)" on page 583
- "GRANT (Table, View, or Nickname Privileges)" on page 590
- "GRANT (Table Space Privileges)" on page 598
- "GRANT (Sequence Privileges)" on page 586
- "GRANT (Routine Privileges)" on page 579

GRANT (Table, View, or Nickname Privileges)

This form of the GRANT statement grants privileges on a table, view, or nickname.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

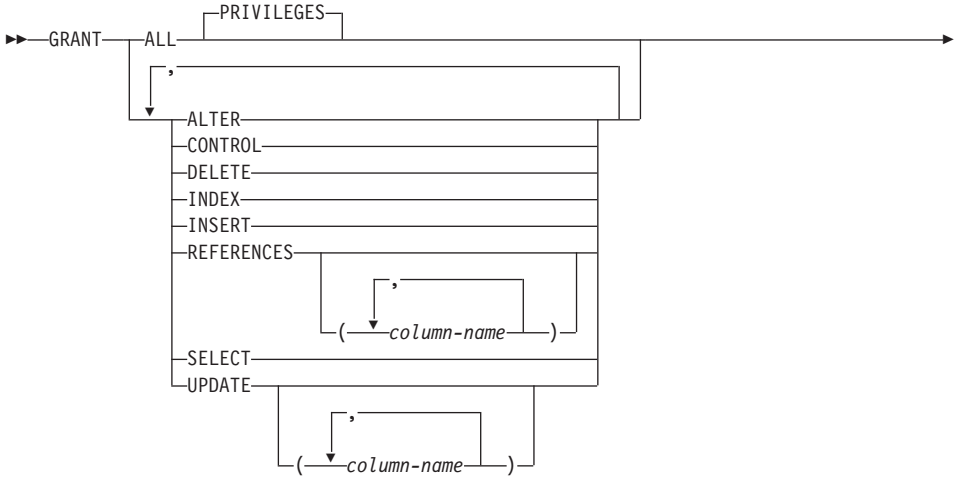
The privileges held by the authorization ID of the statement must include at least one of the following:

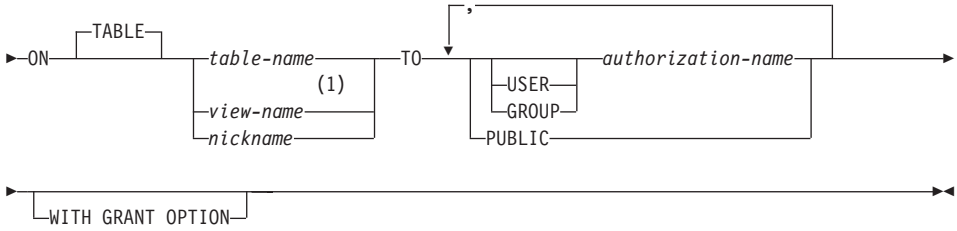
- CONTROL privilege on the referenced table, view, or nickname
- The WITH GRANT OPTION for each identified privilege. If ALL is specified, the authorization ID must have some grantable privilege on the identified table, view, or nickname.
- SYSADM or DBADM authority.

To grant the CONTROL privilege, SYSADM or DBADM authority is required.

To grant privileges on catalog tables and views, either SYSADM or DBADM authority is required.

Syntax:





Notes:

1 ALTER, INDEX, and REFERENCES privileges are not applicable to views.

Description:

ALL or **ALL PRIVILEGES**
Grants all the appropriate privileges, except CONTROL, on the base table, view, or nickname named in the ON clause.

If the authorization ID of the statement has CONTROL privilege on the table, view, or nickname, or DBADM or SYSADM authority, then all the privileges applicable to the object (except CONTROL) are granted. Otherwise, the privileges granted are all those grantable privileges that the authorization ID of the statement has on the identified table, view, or nickname.

If ALL is not specified, one or more of the keywords in the list of privileges must be specified.

ALTER
Grants the privilege to:

- Add columns to a base table definition.
- Create or drop a primary key or unique constraint on a base table.
- Create or drop a foreign key on a base table.

  The REFERENCES privilege on each column of the parent table is also required.
- Create or drop a check constraint on a base table.
- Create a trigger on a base table.
- Add, reset, or drop a column option for a nickname.
- Change a nickname column name or data type.
- Add or change a comment on a base table or a nickname.

CONTROL
Grants:

- All of the appropriate privileges in the list, that is:
  - ALTER, CONTROL, DELETE, INSERT, INDEX, REFERENCES, SELECT, and UPDATE to base tables
  - CONTROL, DELETE, INSERT, SELECT, and UPDATE to views
  - ALTER, CONTROL, INDEX, and REFERENCES to nicknames
- The ability to grant the above privileges (except for CONTROL) to others.
- The ability to drop the base table, view, or nickname.

  This ability cannot be extended to others on the basis of holding CONTROL privilege. The only way that it can be extended is by granting the CONTROL privilege itself and that can only be done by someone with SYSADM or DBADM authority.
- The ability to execute the RUNSTATS utility on the table and indexes.
- The ability to issue the SET INTEGRITY statement against a base table, materialized query table, or staging table.

The definer of a base table, materialized query table, staging table, or nickname automatically receives the CONTROL privilege.

The definer of a view automatically receives the CONTROL privilege if the definer holds the CONTROL privilege on all tables, views, and nicknames identified in the fullselect.

DELETE
Grants the privilege to delete rows from the table or updatable view.

INDEX
Grants the privilege to create an index on a table, or an index specification on a nickname. This privilege cannot be granted on a view. The creator of an index or index specification automatically has the CONTROL privilege on the index or index specification (authorizing the creator to drop the index or index specification). In addition, the creator retains the CONTROL privilege even if the INDEX privilege is revoked.

INSERT
Grants the privilege to insert rows into the table or updatable view and to run the IMPORT utility.

REFERENCES
Grants the privilege to create and drop a foreign key referencing the table as the parent.

If the authorization ID of the statement has one of:

- DBADM or SYSADM authority
- CONTROL privilege on the table
- REFERENCES WITH GRANT OPTION on the table

then the grantee(s) can create referential constraints using all columns of the table as parent key, even those added later using the ALTER TABLE statement. Otherwise, the privileges granted are all those grantable column REFERENCES privileges that the authorization ID of the statement has on the identified table.

The privilege can be granted on a nickname, although foreign keys cannot be defined to reference nicknames.

REFERENCES (*column-name***,...)**
Grants the privilege to create and drop a foreign key 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. Column level REFERENCES privilege cannot be granted on typed tables, typed views, or nicknames (SQLSTATE 42997).

SELECT
Grants the privilege to:

- Retrieve rows from the table or view.
- Create views on the table.
- Run the EXPORT utility against the table or view.

UPDATE
Grants the privilege to use the UPDATE statement on the table or updatable view identified in the ON clause.

If the authorization ID of the statement has one of:

- DBADM or SYSADM authority
- CONTROL privilege on the table or view
- UPDATE WITH GRANT OPTION on the table or view

then the grantee(s) can update all updatable columns of the table or view on which the grantor has with grant privilege as well as those columns added later using the ALTER TABLE statement. Otherwise, the privileges granted are all those grantable column UPDATE privileges that the authorization ID of the statement has on the identified table or view.

UPDATE (*column-name***,...)**
Grants the privilege to use the UPDATE statement to update only those columns specified 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. Column level UPDATE privilege cannot be granted on typed tables, typed views, or nicknames (SQLSTATE 42997).

ON TABLE *table-name* or *view-name* or *nickname*
Specifies the table, view, or nickname on which privileges are to be granted.

No privileges may be granted on an inoperative view or an inoperative materialized query table (SQLSTATE 51024). No privileges may be granted on a declared temporary table (SQLSTATE 42995).

TO
Specifies to whom the privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups. (Previous restrictions on grants to the authorization ID of the user issuing the statement have been removed.)

A privilege granted to a group is not used for authorization checking on static DML statements in a package. Nor is it used when checking authorization on a base table while processing a CREATE VIEW statement.

In DB2 Universal Database, table privileges granted to groups only apply to statements that are dynamically prepared. For example, if the INSERT privilege on the PROJECT table has been granted to group D204 but not UBIQUITY (a member of D204) UBIQUITY could issue the statement:

```
EXEC SQL EXECUTE IMMEDIATE :INSERT_STRING;
```

where the content of the string is:

```
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3114', 'TOOL PROGRAMMING', 'D21', '000260');
```

but could not precompile or bind a program with the statement:

```
EXEC SQL INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3114', 'TOOL PROGRAMMING', 'D21', '000260');
```

PUBLIC
Grants the privileges to all users. (Previous restrictions on the use of privileges granted to PUBLIC for static SQL statements and the CREATE VIEW statement have been removed.)

WITH GRANT OPTION
Allows the specified *authorization-name*s to GRANT the privileges to others.

If the specified privileges include CONTROL, the WITH GRANT OPTION applies to all the applicable privileges except for CONTROL (SQLSTATE 01516).

Rules:

- If neither USER nor GROUP is specified, then
  - If the *authorization-name* is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the *authorization-name* is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.
- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a warning (SQLSTATE 01007) if one or more privileges was not granted. If no privileges were granted, an error is returned (SQLSTATE 42501). (If the package used for processing the statement was precompiled with LANGLEVEL set to SQL92E or MIA, a warning is returned (SQLSTATE 01007), unless the grantor has no privileges on the object of the grant operation.) If CONTROL privilege is specified, privileges will only be granted if the authorization ID of the statement has SYSADM or DBADM authority (SQLSTATE 42501).

Notes:

- ***Compatibilities***
  - For compatibility with DB2 UDB for OS/390 and z/OS:
    - The following syntax is tolerated and ignored:
      - PUBLIC AT ALL LOCATIONS
- Privileges may be granted independently at every level of a table hierarchy. A user with a privilege on a supertable may affect the subtables. For example, an update specifying the supertable *T* may show up as a change to a row in the subtable *S* of *T* done by a user with UPDATE privilege on *T* but without UPDATE privilege on *S*. A user can only operate directly on the subtable if the necessary privilege is held on the subtable.
- Granting nickname privileges has no effect on data source object (table or view) privileges. Typically, data source privileges are required for the table or view that a nickname references when attempting to retrieve data.
- DELETE, INSERT, SELECT, and UPDATE privileges are not defined for nicknames since operations on nicknames depend on the privileges of the authorization ID used at the data source when the statement referencing the nickname is processed.

Examples:

Example 1: Grant all privileges on the table WESTERN_CR to PUBLIC.

```
GRANT ALL ON WESTERN_CR
   TO PUBLIC
```

Example 2: Grant the appropriate privileges on the CALENDAR table so that users 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 USER  PHIL, USER CLAIRE
```

Example 3: Grant all privileges on the COUNCIL table to user FRANK and the ability to extend all privileges to others.

```
GRANT ALL ON COUNCIL
   TO USER FRANK WITH GRANT OPTION
```

Example 4: GRANT SELECT privilege on table CORPDATA.EMPLOYEE to a user named JOHN. There is a user called JOHN and no group called JOHN.

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO JOHN
```

or

```
GRANT SELECT
  ON CORPDATA.EMPLOYEE TO USER JOHN
```

Example 5: GRANT SELECT privilege on table CORPDATA.EMPLOYEE to a group named JOHN. There is a group called JOHN and no user called JOHN.

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO JOHN
```

or

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO GROUP JOHN
```

Example 6: GRANT INSERT and SELECT on table T1 to both a group named D024 and a user named D024.

```
GRANT INSERT, SELECT ON TABLE T1
  TO GROUP D024, USER D024
```

In this case, both the members of the D024 group and the user D024 would be allowed to INSERT into and SELECT from the table T1. Also, there would be two rows added to the SYSCAT.TABAUTH catalog view.

Example 7: GRANT INSERT, SELECT, and CONTROL on the CALENDAR table to user FRANK. FRANK must be able to pass the privileges on to others.

```
GRANT CONTROL ON TABLE CALENDAR
  TO FRANK WITH GRANT OPTION
```

The result of this statement is a warning (SQLSTATE 01516) that CONTROL was not given the WITH GRANT OPTION. Frank now has the ability to grant

any privilege on CALENDAR including INSERT and SELECT as required. FRANK cannot grant CONTROL on CALENDAR to other users unless he has SYSADM or DBADM authority.

Example 8: User JON created a nickname for an Oracle table that had no index. The nickname is ORAREM1. Later, the Oracle DBA defined an index for this table. User SHAWN now wants DB2 to know that this index exists, so that the optimizer can devise strategies to access the table more efficiently. SHAWN can inform DB2 of the index by creating an index specification for ORAREM1. Give SHAWN the index privilege on this nickname, so that he can create the index specification.

```
GRANT INDEX ON NICKNAME ORAREM1
  TO USER SHAWN
```

Related reference:

- "ALTER TABLE" on page 41
- "GRANT (Database Authorities)" on page 569
- "GRANT (Index Privileges)" on page 573
- "GRANT (Package Privileges)" on page 575
- "GRANT (Schema Privileges)" on page 583
- "GRANT (Server Privileges)" on page 588
- "GRANT (Table Space Privileges)" on page 598
- "GRANT (Sequence Privileges)" on page 586
- "GRANT (Routine Privileges)" on page 579

Related samples:

- "tbpriv.sqc -- How to grant, display, and revoke privileges (C)"
- "tbpriv.sqC -- How to grant, display, and revoke privileges (C++)"
- "TbPriv.java -- How to grant, display and revoke privileges on a table (JDBC)"
- "TbPriv.sqlj -- How to grant, display and revoke privileges on a table (SQLj)"

GRANT (Table Space Privileges)

This form of the GRANT statement grants privileges on a table space.

Invocation:

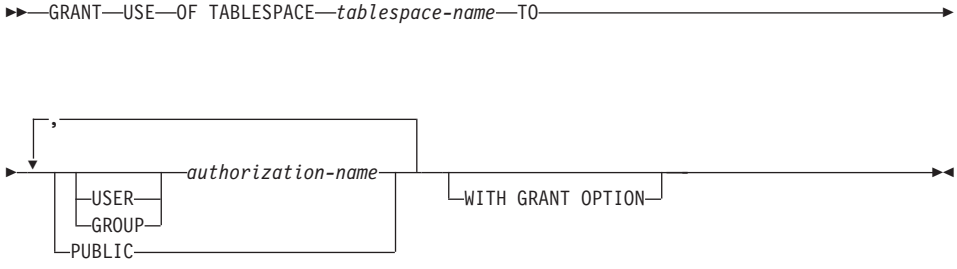
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- WITH GRANT OPTION for use of the table space
- SYSADM, SYSCTRL, or DBADM authority

Syntax:



Description:

USE
Grants the privilege to specify or default to the table space when creating a table. The creator of a table space automatically receives USE privilege with grant option.

OF TABLESPACE *tablespace-name*
Identifies the table space on which the USE privilege is to be granted. The table space cannot be SYSCATSPACE (SQLSTATE 42838) or a system temporary table space (SQLSTATE 42809).

TO
Specifies to whom the USE privilege is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name
Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the USE privilege to all users.

WITH GRANT OPTION
Allows the specified *authorization-name* to GRANT the USE privilege to others.

If the WITH GRANT OPTION is omitted, the specified *authorization-name* can only GRANT the USE privilege to others if they:

- have SYSADM or DBADM authority or
- received the ability to GRANT the USE privilege from some other source.

Notes:

If neither USER nor GROUP is specified, then

- If the *authorization-name* is defined in the operating system only as GROUP, then GROUP is assumed.
- If the *authorization-name* is defined in the operating system only as USER, or if it is undefined, then USER is assumed.
- If the *authorization-name* is defined in the operating system as both, or DCE authentication is used, an error is returned (SQLSTATE 56092).

Examples:

Example 1: Grant user BOBBY the ability to create tables in table space PLANS and to grant this privilege to others.

```
GRANT USE OF TABLESPACE PLANS TO BOBBY WITH GRANT OPTION
```

Related reference:

- "GRANT (Database Authorities)" on page 569
- "GRANT (Index Privileges)" on page 573
- "GRANT (Package Privileges)" on page 575
- "GRANT (Schema Privileges)" on page 583
- "GRANT (Table, View, or Nickname Privileges)" on page 590
- "GRANT (Server Privileges)" on page 588

- "GRANT (Sequence Privileges)" on page 586
- "GRANT (Routine Privileges)" on page 579

INCLUDE

The INCLUDE statement inserts declarations into a source program.

Invocation:

This statement can only be embedded in an application program. It is not an executable statement.

Authorization:

None required.

Syntax:

Description:

SQLCA
Indicates the description of an SQL communication area (SQLCA) is to be included.

SQLDA
Indicates the description of an SQL descriptor area (SQLDA) is to be included.

name
Identifies an external file containing text that is to be included in the source program being precompiled. It may be an SQL identifier without a filename extension or a literal in single quotes (' '). An SQL identifier assumes the filename extension of the source file being precompiled. If a filename extension is not provided by a literal in quotes then none is assumed.

Notes:

- When a program is precompiled, the INCLUDE statement is replaced by source statements. Thus, the INCLUDE statement should be specified at a point in the program such that the resulting source statements are acceptable to the compiler.
- The external source file must be written in the host language specified by the *name*. If it is greater than 18 characters or contains characters not allowed in an SQL identifier then it must be in single quotes. INCLUDE *name* statements may be nested though not cyclical (for example, if A and B

are modules and A contains an INCLUDE *name* statement, then it is not valid for A to call B and then B to call A).

- When the LANGLEVEL precompile option is specified with the SQL92E value, INCLUDE SQLCA should not be specified. SQLSTATE and SQLCODE variables may be defined within the host variable declare section.

Example:

Include an SQLCA in a C program.

```
EXEC SQL INCLUDE SQLCA;

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;
```

Related reference:

- "SQLDA (SQL descriptor area)" in the *SQL Reference, Volume 1*
- "SQLCA (SQL communications area)" in the *SQL Reference, Volume 1*

Related samples:

- "dbcfg.sqc -- Configure database and database manager configuration parameters (C)"
- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"
- "dtformat.sqc -- Load and import data format extensions (C)"
- "tbcreate.sqc -- How to create and drop tables (C)"
- "tbident.sqc -- How to use identity columns (C)"
- "dbcfg.sqC -- Configure database and database manager configuration parameters (C++)"
- "tbcreate.sqC -- How to create and drop tables (C++)"

INSERT

The INSERT statement inserts rows into a table, nickname, or view, or executes the INSTEAD OF insert trigger. Inserting a row into a nickname inserts the row into the data source object to which the nickname refers. Inserting a row into a view also inserts the row into the table on which the view is based, if no INSTEAD OF trigger is defined for the insert operation on this view. If such a trigger is defined, the trigger will be executed instead.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

To execute this statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- INSERT privilege on the table, view or nickname where rows are to be inserted
- CONTROL privilege on the table, view or nickname where rows are to be inserted
- SYSADM or DBADM authority.

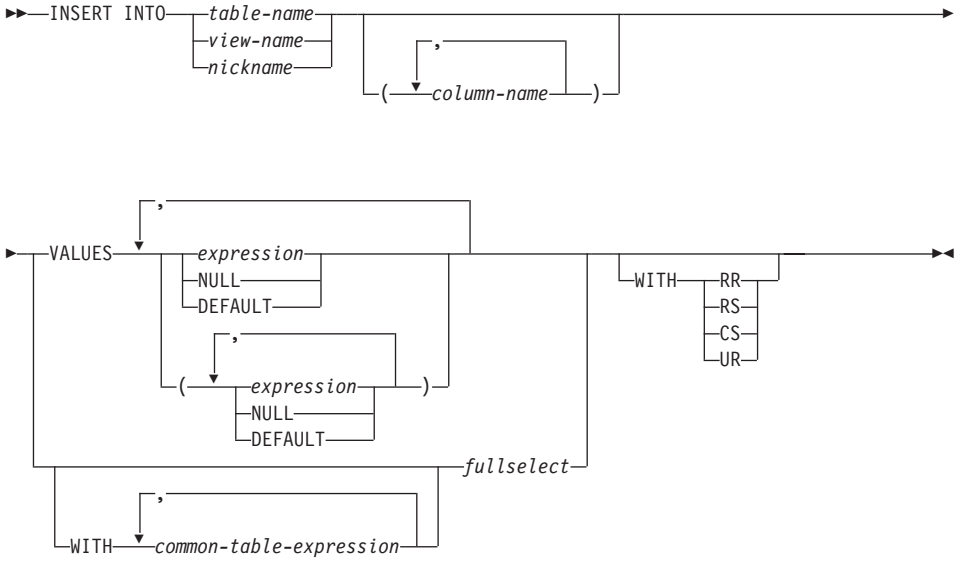
In addition, for each table, view or nickname referenced in any fullselect used in the INSERT statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

GROUP privileges are not checked for static INSERT statements.

If the target of the insert operation is a nickname, the privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges required for the operation on the object at the data source. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

Syntax:



Description:

INTO *table-name*, *view-name* **or** *nickname*
Identifies the object of the insert operation. The name must identify a table, view or nickname that exists at the application server, but it must not identify a catalog table, a system-maintained materialized query table, a view of a catalog table, or a read-only view, unless an INSTEAD OF trigger is defined for the insert operation on the subject view. Rows inserted into a nickname are placed in the data source object to which the nickname refers.

If no INSTEAD OF trigger exists for the insert operation on this view, a value cannot be inserted into a view column that is derived from:

- A constant, expression, or scalar function
- The same base table column as some other column of the view

If 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 these columns.

A row can be inserted into a view that is defined using a UNION ALL if the row satisfies the check constraints of exactly one of the underlying base tables. If a row satisfies the check constraints of more than one table, or no table at all, an error is returned (SQLSTATE 23513).

(*column-name***,...)**
Specifies the columns for which insert values are provided. Each name must be an unqualified name that identifies a column of the table or view.

The same column must not be identified more than once. A view column that cannot accept inserted values (for example, a column based on an expression) must not be identified.

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. 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
Introduces one or more rows of values to be inserted.

Each host variable named must be described in the program in accordance with the rules for declaring host variables.

The number of values for each row must equal the number of names in the column list. The first value is inserted in the first column in the list, the second value in the second column, and so on.

expression
An *expression* can be any expression defined in "Expressions".

NULL
Specifies the null value and should only be specified for nullable columns.

DEFAULT
Specifies that the default value is to be used. The result of specifying DEFAULT depends on how the column was defined, as follows:

- If the column was defined as a generated column based on an expression, the column value is generated by the system, based on that expression.
- If the IDENTITY clause is used, the value is generated by the database manager.
- If the WITH DEFAULT clause is used, the value inserted is as defined for the column (see *default-clause* in "CREATE TABLE").
- If the target of the insert operation is an INSTEAD OF trigger, or the WITH DEFAULT clause, GENERATED clause, and the NOT NULL clause are not used, the inserted value is NULL.
- If the NOT NULL clause is used and the GENERATED clause is not used, or the WITH DEFAULT clause is not used or DEFAULT NULL is used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).
- When inserting into a nickname, the DEFAULT keyword will be passed through the INSERT statement to the data source only if the data source supports the DEFAULT keyword in its query language syntax.

WITH *common-table-expression*
Defines a common table expression for use with the fullselect that follows.

fullselect
Specifies a set of new rows in the form of the result table of a fullselect. There may be one, more than one, or none. If the result table is empty, SQLCODE is set to +100 and SQLSTATE is set to '02000'.

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 in the column list. The value of the first column of the result is inserted in the first column in the list, the second value in the second column, and so on.

WITH
Specifies the isolation level at which the fullselect is executed.

RR
Repeatable Read

RS
Read Stability

CS
Cursor Stability

UR
Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound.

Rules:

- ***Triggers:*** INSERT statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the inserted values. If an insert operation into a view causes an INSTEAD OF trigger to fire, validity, referential integrity, and constraints will be checked against the updates that are performed in the trigger, and not against the view that caused the trigger to fire, or its underlying tables.
- ***Default values:*** The value inserted in any column that is not in the column list is either the default value of the column or null. Columns that do not allow null values and are not defined with NOT NULL WITH DEFAULT must be included in the column list. Similarly, if you insert into a view, the value inserted into any column of the base table that is not in the view is either the default value of the column or null. Hence, all columns of the base table that are not in the view must have either a default value or allow

null values. The only value that can be inserted into a generated column defined with the GENERATED ALWAYS clause is DEFAULT (SQLSTATE 428C9).

- ***Length:*** If the insert value of a column is a number, the column must be a numeric column with the capacity to represent the integral part of the number. If the insert value of a column is a string, the column must either be a string column with a length attribute at least as great as the length of the string, or a datetime column if the string represents a date, time, or timestamp.
- ***Assignment:*** Insert values are assigned to columns in accordance with specific assignment rules.
- ***Validity:*** If the table named, or the base table of the view named, has one or more unique indexes, each row inserted into the table must conform to the constraints imposed by those indexes. If a view whose definition includes WITH CHECK OPTION is named, each row inserted into the view must conform to the definition of the view. For an explanation of the rules governing this situation, see "CREATE VIEW".
- ***Referential Integrity:*** For each constraint defined on a table, each non-null insert value of the foreign key must be equal to a primary key value of the parent table.
- ***Check Constraint:*** Insert values must satisfy the check conditions of the check constraints defined on the table. An INSERT to a table with check constraints defined has the constraint conditions evaluated once for each row that is inserted.
- ***Datalinks:*** Insert statements that include DATALINK values will result in an attempt to link the file if a URL value is included (not empty string or blanks) and the column is defined with FILE LINK CONTROL. Errors in the DATALINK value or in linking the file will cause the insert to fail (SQLSTATE 428D1 or 57050).

Notes:

- After execution of an INSERT statement, the value of the third variable of the SQLERRD(3) portion of the 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. SQLERRD(5) contains the count of all triggered insert, update and delete operations.
- Unless appropriate locks already exist, one or more exclusive locks are acquired at the execution of a successful INSERT statement. Until the locks are released, 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 subselect used in a subquery.

- For further information about locking, see the description of the COMMIT, ROLLBACK, and LOCK TABLE statements.
- If an application is running against a partitioned database, and it is bound with option INSERT BUF, then INSERT with VALUES statements which are not processed using EXECUTE IMMEDIATE may be buffered. DB2 assumes that such an INSERT statement is being processed inside a loop in the application's logic. Rather than execute the statement to completion, it attempts to buffer the new row values in one or more buffers. As a result the actual insertions of the rows into the table are performed later, asynchronous with the application's INSERT logic. Be aware that this asynchronous insertion may cause an error related to an INSERT to be returned on some other SQL statement that follows the INSERT in the application.

  This has the potential to dramatically improve INSERT performance, but is best used with clean data, due to the asynchronous nature of the error handling.
- When a row is inserted into a table that has an identity column, DB2 generates a value for the identity column.
  - For a GENERATED ALWAYS identity column, DB2 always generates the value.
  - For a GENERATED BY DEFAULT column, if a value is not explicitly specified (with a VALUES clause, or subselect), DB2 generates a value.

  The first value generated by DB2 is the value of the START WITH specification for the identity column.
- When a value is inserted for a user-defined distinct type identity column, the entire computation is done in the source type, and the result is cast to the distinct type before the value is actually assigned to the column. (There is no casting of the previous value to the source type prior to the computation.)
- When inserting into a GENERATED ALWAYS identity column, DB2 will always generate a value for the column, and users must not specify a value at insertion time. If a GENERATED ALWAYS identity column is listed in the column-list of the INSERT statement, with a non-DEFAULT value in the VALUES clause, an error occurs (SQLSTATE 428C9).

  For example, assuming that EMPID is defined as an identity column that is GENERATED ALWAYS, then the command:

```
INSERT INTO T2 (EMPID, EMPNAME, EMPADDR)
  VALUES (:hv_valid_emp_id, :hv_name, :hv_addr)
```

  will result in an error.
- When inserting into a GENERATED BY DEFAULT column, DB2 will allow an actual value for the column to be specified within the VALUES clause, or from a subselect. However, when a value is specified in the VALUES clause,

DB2 does not perform any verification of the value. In order to guarantee uniqueness of the values, a unique index on the identity column must be created.

When inserting into a table with a GENERATED BY DEFAULT identity column, without specifying a column list, the VALUES clause can specify the DEFAULT keyword to represent the value for the identity column. DB2 will generate the value for the identity column.

```
INSERT INTO T2 (EMPID, EMPNAME, EMPADDR)
  VALUES (DEFAULT, :hv_name, :hv_addr)
```

In this example, EMPID is defined as an identity column, and thus the value inserted into this column is generated by DB2.

- The rules for inserting into an identity column with a subselect are similar to those for an insert with a VALUES clause. A value for an identity column may only be specified if the identity column is defined as GENERATED BY DEFAULT.

  For example, assume T1 and T2 are tables with the same definition, both containing columns *intcol1* and *identcol2* (both are type INTEGER and the second column has the identity attribute). Consider the following insert:

  ```
  INSERT INTO T2
    SELECT *
    FROM T1
  ```

  This example is logically equivalent to:

  ```
  INSERT INTO T2 (intcol1,identcol2)
    SELECT intcol1, identcol2
    FROM T1
  ```

  In both cases, the INSERT statement is providing an explicit value for the identity column of T2. This explicit specification can be given a value for the identity column, but the identity column in T2 must be defined as GENERATED BY DEFAULT. Otherwise, an error will result (SQLSTATE 428C9).

  If there is a table with a column defined as a GENERATED ALWAYS identity, it is still possible to propagate all other columns from a table with the same definition. For example, given the example tables T1 and T2 described above, the intcol1 values from T1 to T2 can be propagated with the following SQL:

  ```
  INSERT INTO T2 (intcol1)
    SELECT intcol1
    FROM T1
  ```

  Note that, because identcol2 is not specified in the column-list, it will be filled in with its default (generated) value.

- When inserting a row into a single column table where the column is defined as a GENERATED ALWAYS identity column, it is possible to specify a VALUES clause with the DEFAULT keyword. In this case, the application does not provide any value for the table, and DB2 generates the value for the identity column.

  ```
  INSERT INTO IDTABLE
    VALUES(DEFAULT)
  ```

  Assuming the same single column table for which the column has the identity attribute, to insert multiple rows with a single INSERT statement, the following INSERT statement could be used:

  ```
  INSERT INTO IDTABLE
    VALUES (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT)
  ```

- When DB2 generates a value for an identity column, that generated value is consumed; the next time that a value is needed, DB2 will generate a new value. This is true even when an INSERT statement involving an identity column fails or is rolled back.

  For example, assume that a unique index has been created on the identity column. If a duplicate key violation is detected in generating a value for an identity column, an error occurs (SQLSTATE 23505) and the value generated for the identity column is considered to be consumed. This can occur when the identity column is defined as GENERATED BY DEFAULT and the system tries to generate a new value, but the user has explicitly specified values for the identity column in previous INSERT statements. Reissuing the same INSERT statement in this case can lead to success. DB2 will generate the next value for the identity column, and it is possible that this next value will be unique, and that this INSERT statement will be successful.

- If the maximum value for the identity column is exceeded (or minimum value for a descending sequence) in generating a value for an identity column, an error occurs (SQLSTATE 23522). In this situation, the user would have to DROP and CREATE a new table with an identity column having a larger range (that is, change the data type or increment value for the column to allow for a larger range of values).

  For example, an identity column may have been defined with a data type of SMALLINT, and eventually the column runs out of assignable values. To redefine the identity column as INTEGER, the data would need to be unloaded, the table would have to be dropped and recreated with a new definition for the column, and then the data would be reloaded. When the table is redefined, it needs to specify a START WITH value for the identity column such that the next value generated by DB2 will be the next value in the original sequence. To determine the end value, issue a query using MAX of the identity column (for an ascending sequence), or MIN of the identity column (for a descending sequence), before unloading the data.

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: Insert two new departments using one statement into the DEPARTMENT table as in example 2, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
  VALUES ('B11', 'PURCHASING', 'B01'),
         ('E41', 'DATABASE ADMINISTRATION', 'E01')
```

Example 4: Create a temporary table MA_EMP_ACT with the same columns as the EMP_ACT table. Load MA_EMP_ACT with the rows from the EMP_ACT table with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE TABLE MA_EMP_ACT
          ( EMPNO CHAR(6)  NOT NULL,
            PROJNO CHAR(6)  NOT NULL,
            ACTNO SMALLINT  NOT NULL,
            EMPTIME DEC(5,2),
            EMSTDATE DATE,
            EMENDATE  DATE )
INSERT INTO MA_EMP_ACT
  SELECT * FROM EMP_ACT
    WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 5: Use a C program statement to add a skeleton project to the PROJECT table. 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.

```
EXEC SQL  INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
             VALUES (:PRJNO, :PRJNM, :DPTNO, :REMP, CURRENT DATE);
```

Related concepts:

- "Queries" in the *SQL Reference, Volume 1*

Related reference:

- "Expressions" in the *SQL Reference, Volume 1*
- "CREATE TABLE" on page 332
- "CREATE VIEW" on page 463
- "Assignments and comparisons" in the *SQL Reference, Volume 1*

Related samples:

- "dtlob.sqc -- How to use the LOB data type (C)"
- "tbident.sqc -- How to use identity columns (C)"
- "tbmod.sqc -- How to modify table data (C)"
- "tbtrig.sqc -- How to use a trigger on a table (C)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "tbmod.sqC -- How to modify table data (C++)"
- "tbtrig.sqC -- How to use a trigger on a table (C++)"
- "DtLob.java -- How to use LOB data type (JDBC)"
- "TbIdent.java -- How to use Identity Columns (JDBC)"
- "TbMod.java -- How to modify table data (JDBC)"
- "TbTrig.java -- How to use triggers (JDBC)"
- "TbIdent.sqlj -- How to use Identity Columns (SQLj)"
- "TbMod.sqlj -- How to modify table data (SQLj)"
- "TbTrig.sqlj -- How to use triggers (SQLj)"
- "updat.sqb -- How to update, delete and insert table data (MF COBOL)"

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:

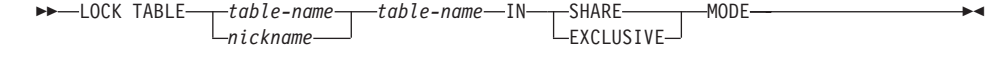
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SELECT privilege on the table
- CONTROL privilege on the table
- SYSADM or DBADM authority.

Syntax:



Description:

table-name **or** *nickname*
Identifies the table or nickname. The *table-name* must identify a table that exists at the application server, but it must not identify a catalog table. It cannot be a declared temporary table (SQLSTATE 42995). If the *table-name* is a typed table, it must be the root table of the table hierarchy (SQLSTATE 428DR). When a nickname is specified, DB2 will lock the underlying object (i.e., a table or view) of the data source to which the nickname refers.

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. Note that EXCLUSIVE MODE does not prevent concurrent application processes that are running at isolation level Uncommitted Read (UR) from executing read-only operations on the table.

Notes:

- 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 at least until the termination of the unit of work.
- In a partitioned database, a table lock is first acquired at the first partition in the database partition group (the partition with the lowest number) and then at other partitions. If the LOCK TABLE statement is interrupted, the table may be locked on some partitions but not on others. If this occurs, either issue another LOCK TABLE statement to complete the locking on all partitions, or issue a COMMIT or ROLLBACK statement to release the current locks.
- This statement affects all partitions in the database partition group.

Example:

Obtain a lock on the table EMP. Do not allow other programs either to read or update the table.

```
LOCK TABLE EMP IN EXCLUSIVE MODE
```

OPEN

The OPEN statement opens a cursor so that it can be used to fetch rows from its result table.

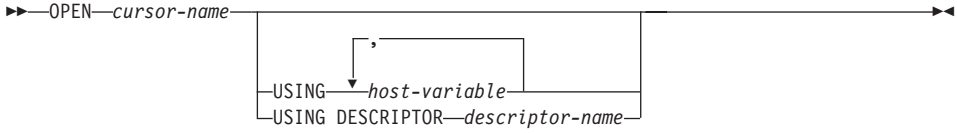
Invocation:

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:

For the authorization required to use a cursor, see "DECLARE CURSOR".

Syntax:



Description:

cursor-name
Names a cursor that is defined in a DECLARE CURSOR statement that was stated earlier in the program. When the OPEN statement is executed, the cursor must be in the closed state.

The DECLARE CURSOR statement must identify a SELECT statement, in one of the following ways:

- Including the SELECT statement in the DECLARE CURSOR statement
- Including a *statement-name* that names a prepared SELECT statement.

The result table of the cursor is derived by evaluating the SELECT statement. The evaluation uses the current values of any special registers or PREVVAL expressions specified in the SELECT statement, and the current values of any host variables specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table may be derived during the execution of the OPEN statement, and a temporary table may be created to hold them; or they may 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".

USING
Introduces a list of host variables whose values are substituted for the parameter markers (question marks) of a prepared statement. If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, USING must be used. If the prepared statement does not include parameter markers, USING is ignored.

host-variable
Identifies a variable described in the program in accordance with the rules for declaring host 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. Where appropriate, locator variables and file reference variables can be provided as the source of values for parameter markers.

DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of host variables.

Before the OPEN statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables.

The 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 LOB result columns need to be accommodated, there must be two SQLVAR entries for every select-list item (or column of the result table).

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

Rules:

- When the SELECT statement of the cursor is evaluated, each parameter marker in the statement is effectively replaced by its corresponding host variable. 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.

- Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:
  - V must be compatible with the target.
  - If V is a string, its length must not be greater than the length attribute of the target.
  - If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
  - If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.

  When the SELECT statement of the cursor is evaluated, the value used in place of P is the value of the target variable for P. For example, if V is CHAR(6), and the target is CHAR(8), the value used in place of P is the value of V padded with two blanks.
- The USING clause is intended for a prepared SELECT statement that contains parameter markers. However, it can also be used when the SELECT statement of the cursor is part of the DECLARE CURSOR statement. In this case the OPEN statement is executed as if each host 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 host variables in the SELECT statement. The effect is to override the values of the host variables in the SELECT statement of the cursor with the values of the host variables specified in the USING clause.

Notes:

- ***Closed state of cursors:*** All cursors in a program are in the closed state when the program is initiated and when it initiates a ROLLBACK statement.

  All cursors, except open cursors declared WITH HOLD, are in a closed state when a program issues a COMMIT statement.

  A cursor can also be in the closed state because a CLOSE statement was executed or an error was detected that made the position of the cursor unpredictable.
- To retrieve rows from the result table of a cursor, execute a FETCH statement 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:*** In some cases, the result table of a cursor is derived during the execution of FETCH statements. In other cases, the

temporary table method is used instead. With this method the entire result table is transferred to a temporary table during the execution of the OPEN statement. When a temporary table is used, the results of a program can differ in these ways:

- An error can occur during OPEN that would otherwise not occur until some later FETCH statement.
- INSERT, UPDATE, and DELETE statements executed in the same transaction while the cursor is open cannot affect the result table.
- Any NEXTVAL expressions in the SELECT statement are evaluated 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 if issued from the same unit of work, and any NEXTVAL expressions in the SELECT statement are evaluated as each row is fetched. This result table can also be affected by operations executed by the same unit of work, and the effect of such operations is not always predictable. For example, if cursor C is positioned on a row of its result table defined as SELECT * FROM T, and a new row is inserted into T, the effect of that insert on the result table is not predictable because its rows are not ordered. Thus a subsequent FETCH C may or may not retrieve the new row of T.

- Statement caching affects cursors declared open by the OPEN statement.

Examples:

Example 1: Write the embedded statements in a COBOL program that will:

1. Define a cursor C1 that is to be used to retrieve all rows from the DEPARTMENT table for departments that are administered by (ADMRDEPT) department 'A00'.
2. Place the cursor C1 before the first row to be fetched.

```
EXEC SQL  DECLARE C1 CURSOR FOR
                SELECT DEPTNO, DEPTNAME, MGRNO
                  FROM DEPARTMENT
                  WHERE ADMRDEPT = 'A00'
END-EXEC.


EXEC SQL  OPEN C1
END-EXEC.
```

Example 2: Code an OPEN statement to associate a cursor DYN_CURSOR with a dynamically defined select-statement in a C program. Assuming two parameter markers are used in the predicate of the select-statement, two host variable references are supplied with the OPEN statement to pass integer and

varchar(64) values between the application and the database. (The related host variable definitions, PREPARE statement, and DECLARE CURSOR statement are also shown in the example below.)

```
EXEC SQL  BEGIN DECLARE SECTION;
  static short    hv_int;
  char            hv_vchar64[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;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING DESCRIPTOR :sqlda;
```

Related reference:

- "DECLARE CURSOR" on page 482
- "EXECUTE" on page 544
- "PREPARE" on page 620
- "SQLDA (SQL descriptor area)" in the *SQL Reference, Volume 1*

Related samples:

- "dynamic.sqb -- How to update table data with cursor dynamically (MF COBOL)"
- "spserver.sqc -- A variety of types of stored procedures (C)"
- "tut_read.sqc -- How to read tables (C)"
- "udfemsrv.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"
- "spserver.sqC -- A variety of types of stored procedures (C++)"
- "tut_read.sqC -- How to read tables (C++)"
- "udfemsrv.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)"

PREPARE

The PREPARE statement is used by application programs to dynamically prepare an SQL statement for execution. The PREPARE statement creates an executable SQL statement, called a *prepared statement*, from a character string form of the statement, called a *statement string*.

Invocation:

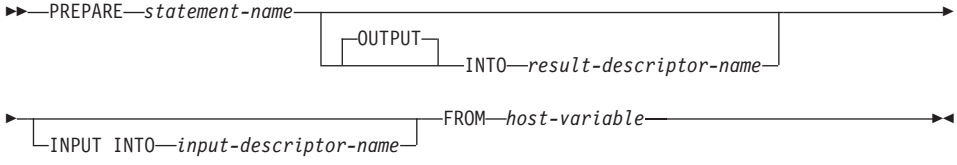
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization:

For statements where authorization checking is performed at statement preparation time (DML), the privileges held by the authorization ID of the statement must include those required to execute the SQL statement specified by the PREPARE statement. The authorization ID of the statement may be affected by the bind option DYNAMICRULES.

For statements where authorization checking is performed at statement execution (DDL, GRANT, and REVOKE statements), no authorization is required to use the statement; however, the authorization is checked when the prepared statement is executed.

Syntax:



Description:

statement-name
Names the prepared statement. If the name identifies an existing prepared statement, that previously prepared statement is destroyed. The name must not identify a prepared statement that is the SELECT statement of an open cursor.

OUTPUT INTO
If OUTPUT INTO is used, and the PREPARE statement executes successfully, information about the output parameter markers in the prepared statement is placed in the SQLDA specified by *result-descriptor-name*.

result-descriptor-name
Specifies the name of an SQLDA. (The DESCRIBE statement may be used as an alternative to this clause.)

INPUT INTO
If INPUT INTO is used, and the PREPARE statement executes successfully, information about the input parameter markers in the prepared statement is placed in the SQLDA specified by *input-descriptor-name*. Input parameter markers are always considered nullable, regardless of usage.

input-descriptor-name
Specifies the name of an SQLDA. (The DESCRIBE statement may be used as an alternative to this clause.)

FROM
Introduces the statement string. The statement string is the value of the specified host variable.

host-variable
Specifies a host variable that is described in the program in accordance with the rules for declaring character string variables. It must be a fixed-length or varying-length character-string variable.

Rules:

- ***Rules for statement strings:*** The statement string must be an executable statement that can be dynamically prepared. It must be one of the following SQL statements:
  - ALTER
  - CALL
  - COMMENT
  - COMMIT
  - CREATE
  - DECLARE GLOBAL TEMPORARY TABLE
  - DELETE
  - DROP
  - EXPLAIN
  - FLUSH EVENT MONITOR
  - FLUSH PACKAGE CACHE
  - GRANT
  - INSERT
  - LOCK TABLE
  - REFRESH TABLE
  - RELEASE SAVEPOINT

- RENAME TABLE
- RENAME TABLESPACE
- REVOKE
- ROLLBACK
- SAVEPOINT
- select-statement
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET SCHEMA
- SET SERVER OPTION
- UPDATE

- ***Parameter markers:*** Although a statement string cannot include references to host variables, it may include *parameter markers*. These can be replaced by the values of host variables when the prepared statement is executed. In the case of a CALL statement, a parameter marker can also be used for OUT and INOUT arguments to the stored procedure. After the CALL is executed, the returned value for the argument will be assigned to the host variable corresponding to the parameter marker.

  A parameter marker is a question mark (?) that is used where a host 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" and "EXECUTE".

  There are two types of parameter markers:

  Typed parameter marker
  A parameter marker that is specified along with its target data type. It has the general form:

  ```
  CAST(? AS data-type)
  ```

This notation is not a function call, but a "promise" that the type of the parameter at run time will be of the data type specified or some data type that can be converted to the specified data type. For example, in:

```
  UPDATE EMPLOYEE
SET LASTNAME = TRANSLATE(CAST(? AS VARCHAR(12)))
WHERE EMPNO = ?
```

the value of the argument of the TRANSLATE function will be provided at run time. The data type of that value will either be VARCHAR(12), or some type that can be converted to VARCHAR(12).

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 host variable is supported and the data type is based on the promise made in the CAST function.

Untyped parameters markers can be used in dynamic SQL statements in selected locations where host variables are supported. These locations and the resulting data type are found in Table 12. The locations are grouped in this table into expressions, predicates, built-in functions, and user-defined routines to assist in determining applicability of an untyped parameter marker. When an untyped parameter marker is used in a function (including arithmetic operators, CONCAT and datetime operators) with an unqualified function name, the qualifier is set to 'SYSIBM' for the purposes of function resolution.

Table 12. Untyped Parameter Marker Usage

Untyped Parameter Marker Location	Data Type
Expressions (including select list, CASE, and VALUES)	
Alone in a select list	Error
Both operands of a single arithmetic operator, after considering operator precedence and order of operation rules. Includes cases such as: `? + ? + 10`	Error

Table 12. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
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.
Labelled duration within a datetime expression. (Note that the portion of a labelled duration that indicates the type of units cannot be a parameter marker.)	DECIMAL(15,0)
Any other operand of a datetime expression (for instance 'timecol + ?' or '? - datecol').	Error
Both operands of a CONCAT operator	Error
One operand of a CONCAT operator where the other operand is a non-CLOB character data type	If one operand is either CHAR(n) or VARCHAR(n), where n is less than 128, then other is VARCHAR(254 - n). In all other cases the data type is VARCHAR(254).
One operand of a CONCAT operator where the other operand is a non-DBCLOB graphic data type.	If one operand is either GRAPHIC(n) or VARGRAPHIC(n), where n is less than 64, then other is VARCHAR(127 - n). In all other cases the data type is VARCHAR(127).
One operand of a CONCAT operator where the other operand is a large object string.	Same as that of the other operand.
As a value on the right hand side of a SET clause of an UPDATE statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, then it is the structured type, also indicating the returns type of the transform function.
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

Table 12. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
Any or all expressions following WHEN in a simple CASE expression.	Result of applying the Rules for Result Data Types 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 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.	Error
Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the column-expressions in the same position in all other row-expressions are untyped parameter markers.	Error
Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the expression in the same position of at least one other row-expression is not an untyped parameter marker or NULL.	Result of applying the Rules for Result Data Types on all operands that are other than untyped parameter markers.
Alone as a column-expression in a single-row VALUES clause within an INSERT statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, then it is the structured type, also indicating the returns type of the transform function.
Alone as a column-expression in a multi-row VALUES clause within an INSERT statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, then it is the structured type, also indicating the returns type of the transform function.
As a value on the right side of a SET special register statement	The data type of the special register.
Predicates	

Table 12. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
Both operands of a comparison operator	Error
One operand of a comparison operator where the other operand other than an untyped parameter marker.	The data type of the other operand
All operands of a BETWEEN predicate	Error
Either 1st and 2nd, or 1st and 3rd operands of a BETWEEN predicate	Same as that of the only non-parameter marker.
Remaining BETWEEN situations (i.e. one untyped parameter marker only)	Result of applying the Rules for Result Data Types on all operands that are other than untyped parameter markers.
All operands of an IN predicate	Error
The 1st operand of an IN predicate where the right hand side is not a subselect; for example, ? IN (?,A,B), or ? IN (A,?,B,?).	Result of applying the Rules for Result Data Types on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers.
The 1st operand of an IN predicate where the right hand side is a fullselect.	Data type of the selected column
Any or all operands of the IN list of the IN predicate	Results of applying the Rules for Result Data Types on all operands of the IN predicate (operands to the left and right of the IN predicate) that are other than untyped parameter markers.
All three operands of the LIKE predicate.	Match expression (operand 1) and pattern expression (operand 2) are VARCHAR(32672). Escape expression (operand 3) is VARCHAR(2).
The match expression of the LIKE predicate when either the pattern expression or the escape expression is other than an untyped parameter marker.	Either VARCHAR(32672) or VARGRAPHIC(16336) depending on the data type of the first operand that is not an untyped parameter marker.
The pattern expression of the LIKE predicate when either the match expression or the escape expression is other than an untyped parameter marker.	Either VARCHAR(32672) or VARGRAPHIC(16336) depending on the data type of the first operand that is not an untyped parameter marker. If the data type of the match expression is BLOB, the data type of the pattern expression is assumed to be BLOB(32672).

Table 12. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
The escape expression of the LIKE predicate when either the match expression or the pattern expression is other than an untyped parameter marker.	Either VARCHAR(2) or VARGRAPHIC(1) depending on the data type of the first operand that is not an untyped parameter marker. If the data type of the match expression or pattern expression is BLOB, the data type of the escape expression is assumed to be BLOB(1).
Operand of the NULL predicate	error
Built-in Functions	
All operands of COALESCE (also called VALUE) or NULLIF	Error
Any operand of COALESCE or NULLIF where at least the first operand is other than an untyped parameter marker.	Result of applying the Rules for Result Data Types on all operands that are other than untyped parameter markers.
POSSTR (both operands)	Both operands are VARCHAR(32672).
POSSTR (one operand where the other operand is a character data type).	VARCHAR(32672).
POSSTR (one operand where the other operand is a graphic data type).	VARGRAPHIC(16336).
POSSTR (the search-string operand when the other operand is a BLOB).	BLOB(32672).
SUBSTR (1st operand)	VARCHAR(32672)
SUBSTR (2nd and 3rd operands)	INTEGER
The 1st operand of the TRANSLATE scalar function.	Error
The 2nd and 3rd operands of the TRANSLATE scalar function.	VARCHAR(32672) if the first operand is a character type. VARGRAPHIC(16336) if the first operand is a graphic type.
The 4th operand of the TRANSLATE scalar function.	VARCHAR(1) if the first operand is a character type. VARGRAPHIC(1) if the first operand is a graphic type.
The 2nd operand of the TIMESTAMP scalar function.	TIME
Unary minus	DOUBLE-PRECISION
Unary plus	DOUBLE-PRECISION
First operand of the VARCHAR_FORMAT function.	TIMESTAMP

Table 12. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
First operand of the TIMESTAMP_FORMAT function.	VARCHAR (length of a short string)
All other operands of all other scalar functions.	Error
Operand of a column function.	Error
User-defined Routines	
Argument of a function	Error
Argument of a method	Error
Argument of a procedure	The data type of the parameter, as defined when the procedure was created.

Notes:

- When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is invalid, the error condition is reported in the SQLCA. Any subsequent EXECUTE or OPEN statement that references this statement will also receive the same error (due to an implicit prepare done by the system) unless the error has been corrected.
- Prepared statements can be referred to in the following kinds of statements, with the restrictions shown:

In...	The prepared statement...
DESCRIBE	can be any statement
DECLARE CURSOR	must be SELECT
EXECUTE	must *not* be SELECT

- A prepared statement can be executed many times. Indeed, 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.
- Statement caching affects repeated preparations.

Examples:

Example 1: Prepare and execute a non-select-statement in a COBOL program. Assume the statement is contained in a host variable HOLDER and that the program will place a statement string into the host variable based on some instructions from the user. The statement to be prepared does not have any parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER
END-EXEC.
EXEC SQL  EXECUTE STMT_NAME
END-EXEC.
```

Example 2: Prepare and execute a non-select-statement as in example 1, except code it for a C program. Also assume the statement to be prepared can contain any number of parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :holder;
EXEC SQL  EXECUTE STMT_NAME USING DESCRIPTOR :insert_da;
```

Assume that the following statement is to be prepared:

```
INSERT INTO DEPT VALUES(?, ?, ?, ?)
```

The columns in the DEPT table are defined as follows:

```
DEPT_NO   CHAR(3) NOT NULL, -- department number
DEPTNAME  VARCHAR(29), -- department name
MGRNO     CHAR(6), -- manager number
ADMRDEPT  CHAR(3)  -- admin department number
```

To insert department number G01 named COMPLAINTS, which has no manager and reports to department A00, the structure INSERT_DA should have the values in Table 13 before issuing the EXECUTE statement.

Table 13.

SQLDA field	**Value**
SQLDAID	SQLDA
SQLDABC	192 (See note 1.)
SQLN	4
SQLD	4
SQLTYPE	452
SQLLEN	3
SQLDATA	*pointer to* G01
SQLIND	(See note 2.)
SQLNAME	
SQLTYPE	449
SQLLEN	29
SQLDATA	*pointer to* COMPLAINTS
SQLIND	*pointer to* 0

Table 13. (continued)

SQLDA field	Value
SQLNAME	
SQLTYPE	453
SQLLEN	6
SQLDATA	(See note 3.)
SQLIND	*pointer to* -1
SQLNAME	
SQLTYPE	453
SQLLEN	3
SQLDATA	*pointer to* A00
SQLIND	*pointer to* 0
SQLNAME	

Notes:

1. This value is for a PREPARE done from a 32-bit application. If the PREPARE was done in a 64-bit application, then SQLDABC would have the value 240.
2. The value in SQLIND for this SQLVAR is ignored because the SQLTYPE identifies a non-nullable data type.
3. The value in SQLDATA for this SQLVAR is ignored because the value of SQLIND indicates this is a NULL value.

Related reference:

- "Identifiers" in the *SQL Reference, Volume 1*
- "LIKE predicate" in the *SQL Reference, Volume 1*
- "DESCRIBE" on page 504
- "EXECUTE" on page 544
- "OPEN" on page 615
- "Rules for result data types" in the *SQL Reference, Volume 1*

Related samples:

- "inpsrv.sqb -- Demonstrates stored procedures using the SQLDA structure (MF COBOL)"
- "trigsql.sqb -- How to use a trigger on a table (MF COBOL)"
- "tbread.sqc -- How to read tables (C)"
- "tut_use.sqc -- How to modify a database (C)"

- “tbread.sqC -- How to read tables (C++)”
- “tut_use.sqC -- How to modify a database (C++)”

REFRESH TABLE

The REFRESH TABLE statement refreshes the data in a materialized query table.

Invocation:

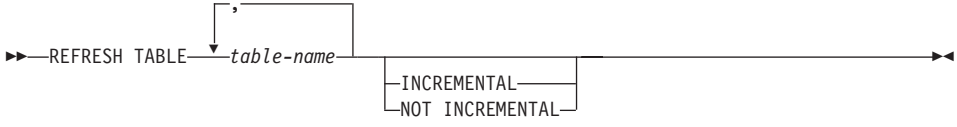
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CONTROL privilege on the table.

Syntax:



Description:

table-name
Identifies the table to be refreshed.

The name, including the implicit or explicit schema, must identify a table that already exists at the current server. The table must allow the REFRESH TABLE statement (SQLSTATE 42809). This includes materialized query tables defined with:

- REFRESH IMMEDIATE
- REFRESH DEFERRED

INCREMENTAL
Specifies an incremental refresh for the table by considering only the appended portion (if any) of its underlying tables or the content of an associated staging table (if one exists and its contents are consistent). If such a request cannot be satisfied (that is, the system detects that the materialized query table definition needs to be fully recomputed), an error (SQLSTATE 55019) is returned.

NOT INCREMENTAL
Specifies a full refresh for the table by recomputing the materialized query table definition.

If neither INCREMENTAL nor NOT INCREMENTAL is specified, the system will determine whether incremental processing is possible; if not, full refresh will be performed. If a staging table is present for the materialized query table that is to be refreshed, and incremental processing is not possible because the staging table is in a pending state, an error is returned (SQLSTATE 428A8). Full refresh will be performed if the staging table or the materialized query table is in an inconsistent state; otherwise, the contents of the staging table will be used for incremental processing.

Notes:

- If REFRESH TABLE is issued on a materialized query table that references one or more nicknames, the issuer must have authority to select from the remote tables (SQLSTATE 42501).
- When the statement is used to refresh a REFRESH IMMEDIATE materialized query table whose underlying tables have been loaded, the system may choose to incrementally refresh the materialized query table with the appended portions of its underlying tables. When the statement is used to refresh a REFRESH DEFERRED materialized query table with a supporting staging table, the system may choose to incrementally refresh the materialized query table with the appended portions of its underlying tables that have been captured in the staging table. However, there are some situations in which this optimization is not possible, and a full refresh (that is, a recomputation of the materialized query table definition) is necessary to ensure data integrity. The user can explicitly request incremental maintenance by specifying the INCREMENTAL option; if this optimization is not possible, the system returns an error. There are certain scenarios in which the system will use incremental processing while placing the responsibility for consistency on the user.
- If the materialized query table has an associated staging table, the staging table is pruned when the refresh is successfully performed.

Related reference:

- “SET INTEGRITY” on page 704
- “CREATE USER MAPPING” on page 461

RELEASE (Connection)

The RELEASE (Connection) 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.

Authorization:

None Required.

Syntax:

Notes:

1 Note that an application server named CURRENT or ALL can only be identified by a host variable or a delimited identifier.

Description:

server-name or *host-variable*
Identifies the application server by the specified *server-name* or a *host-variable* which contains the *server-name*.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-justified and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The specified database-alias or the database-alias contained in the host variable must identify an existing connection of the application process. If the database-alias does not identify an existing connection, an error (SQLSTATE 08003) is raised.

CURRENT
Identifies the current connection of the application process. The application process must be in the connected state. If not, an error (SQLSTATE 08003) is raised.

ALL or ALL SQL
Identifies all existing connections of the application process. This form of the RELEASE statement places all existing connections of the application process in the release-pending state. All connections will therefore be destroyed during the next commit operation. An error or warning does not occur if no connections exist when the statement is executed.

Examples:

Example 1: The SQL connection to IBMSTHDB is no longer needed by the application. The following statement will cause it to be destroyed during the next commit operation:

```
EXEC SQL RELEASE IBMSTHDB;
```

Example 2: The current connection is no longer needed by the application. The following statement will cause it to be destroyed during the next commit operation:

```
EXEC SQL RELEASE CURRENT;
```

Example 3: If an application has no need to access the databases after a commit but will continue to run for a while, then it is better not to tie up those connections unnecessarily. The following statement can be executed before the commit to ensure all connections will be destroyed at the commit:

```
EXEC SQL RELEASE ALL;
```

RELEASE SAVEPOINT

The RELEASE SAVEPOINT statement is used to indicate that the application no longer wishes to have the named savepoint maintained. After this statement has been invoked, rollback to the savepoint is no longer possible.

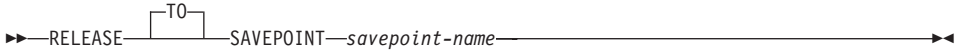
Invocation:

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
The named savepoint is released. Rollback to that savepoint is no longer possible. If the named savepoint does not exist, an error is issued (SQLSTATE 3B001).

Notes:

- The name of the savepoint that was released can now 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.

Example:

Example 1: Release a savepoint named SAVEPOINT1.

```
RELEASE SAVEPOINT SAVEPOINT1
```

RENAME

The RENAME statement renames an existing table or index.

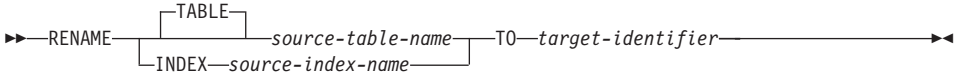
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include either SYSADM or DBADM authority, CONTROL privilege on the table or index, or ALTERIN privilege on the schema.

Syntax:



Description:

TABLE *source-table-name*
Names the existing table that is to be renamed. The name, including the schema name, must identify a table that already exists in the database (SQLSTATE 42704). It must not be the name of a catalog table (SQLSTATE 42832), a materialized query table, a typed table (SQLSTATE 42997), a declared global temporary table (SQLSTATE 42995), a nickname, or an object other than a table or an alias (SQLSTATE 42809). The TABLE keyword is optional.

INDEX *source-index-name*
Names the existing index that is to be renamed. The name, including the schema name, must identify an index that already exists in the database (SQLSTATE 42704). It must not be the name of an index on a declared global temporary table (SQLSTATE 42995). The schema name must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42832).

target-identifier
Specifies the new name for the table or index without a schema name. The schema name of the source object is used to qualify the new name for the object. The qualified name must *not* identify a table, view, alias, or index that already exists in the database (SQLSTATE 42710).

Rules:

When renaming a table, the source table must not:

- Be referenced in any existing view definitions or materialized query table definitions
- Be referenced in any triggered SQL statements in existing triggers or be the subject table of an existing trigger
- Be referenced in an SQL function
- Have any check constraints
- Have any generated columns other than the identity column
- Be a parent or dependent table in any referential integrity constraints
- Be the scope of any existing reference column.

An error (SQLSTATE 42986) is returned if the source table violates one or more of these conditions.

When renaming an index:

- The source index must not be a system-generated index for an implementation table on which a typed table is based (SQLSTATE 42858).

Notes:

- Catalog entries are updated to reflect the new table or index name.
- *All* authorizations associated with the source table or index name are *transferred* to the new table or index name (the authorization catalog tables are updated appropriately).
- Indexes defined over the source table are *transferred* to the new table (the index catalog tables are updated appropriately).
- RENAME TABLE invalidates any packages that are dependent on the source table. RENAME INDEX invalidates any packages that are dependent on the source index.
- If an alias is used for the *source-table-name*, it must resolve to a table name. The table is renamed within the schema of this table. The alias is not changed by the RENAME statement and continues to refer to the old table name.
- A table with primary key or unique constraints can be renamed if none of the primary key or unique constraints are referenced by any foreign key.

Examples:

Change the name of the EMP table to EMPLOYEE.

```
RENAME TABLE EMP TO EMPLOYEE
RENAME TABLE ABC.EMP TO EMPLOYEE
```

Change the name of the index NEW-IND to IND.

```
RENAME INDEX NEW-IND TO IND
RENAME INDEX ABC.NEW-IND TO IND
```

RENAME TABLESPACE

The RENAME TABLESPACE statement renames an existing table space.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include either SYSADM or SYSCTRL authority.

Syntax:

Description:

source-tablespace-name
Specifies the existing table space that is to be renamed, as a one-part name. It is an SQL identifier (either ordinary or delimited). The table space name must identify a table space that already exists in the catalog (SQLSTATE 42704).

target-tablespace-name
Specifies the new name for the table space, as a one-part name. It is an SQL identifier (either ordinary or delimited). The new table space name must *not* identify a table space that already exists in the catalog (SQLSTATE 42710), and it cannot start with 'SYS' (SQLSTATE 42939).

Rules:

- The SYSCATSPACE table space cannot be renamed (SQLSTATE 42832).
- Any table spaces with "rollforward pending" or "rollforward in progress" states cannot be renamed (SQLSTATE 55039)

Notes:

- Renaming a table space will update the minimum recovery time of a table space to the point in time when the rename took place. This implies that a roll forward at the table space level must be to at least this point in time.
- The new table space name must be used when restoring a table space from a backup image, where the rename was done after the backup was created.

Example:

Change the name of the table space USERSPACE1 to DATA2000:

```
RENAME TABLESPACE USERSPACE1 TO DATA2000
```

REVOKE (Database Authorities)

This form of the REVOKE statement revokes authorities that apply to the entire database.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

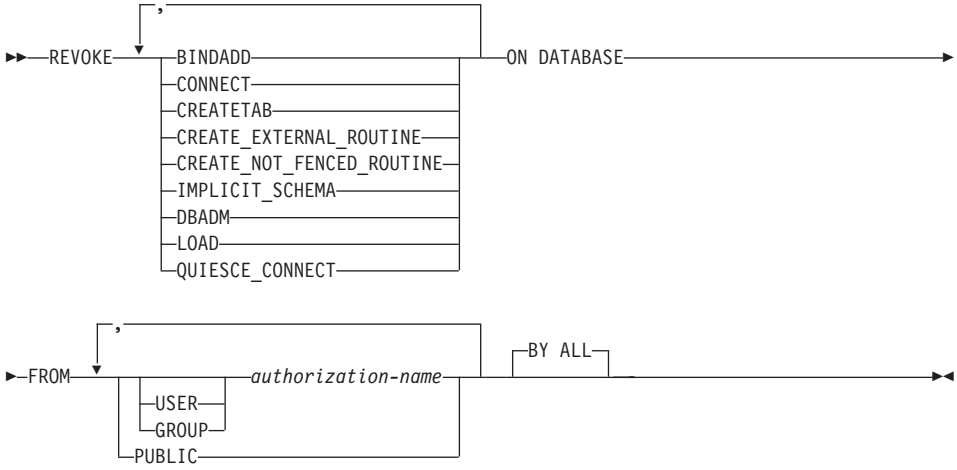
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- DBADM authority
- SYSADM authority

To revoke DBADM authority, SYSADM authority is required.

Syntax:



Description:

BINDADD
Revokes the authority to create packages. The creator of a package automatically has the CONTROL privilege on that package and retains this privilege even if his BINDADD authority is subsequently revoked.

The BINDADD authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority.

CONNECT
Revokes the authority to access the database.

Revoking the CONNECT authority from a user does not affect any privileges that were granted to that user on objects in the database. If the user is subsequently granted the CONNECT authority again, all previously held privileges are still valid (assuming they were not explicitly revoked).

The CONNECT authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

CREATETAB
Revokes the authority to create tables. The creator of a table automatically has the CONTROL privilege on that table, and retains this privilege even if his CREATETAB authority is subsequently revoked.

The CREATETAB authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

CREATE_EXTERNAL_ROUTINE
Revokes the authority to register external routines. Once an external routine has been registered, it continues to exist, even if CREATE_EXTERNAL_ROUTINE is subsequently revoked from the authorization ID that registered the routine.

CREATE_EXTERNAL_ROUTINE authority cannot be revoked from an *authorization-name* holding DBADM or CREATE_NOT_FENCED_ROUTINE authority without also revoking DBADM or CREATE_NOT_FENCED_ROUTINE authority (SQLSTATE 42504).

CREATE_NOT_FENCED_ROUTINE
Revokes the authority to register routines that execute in the database manager's process. Once a routine has been registered as not fenced, it continues to run in this manner, even if CREATE_NOT_FENCED_ROUTINE is subsequently revoked from the authorization ID that registered the routine.

CREATE_NOT_FENCED_ROUTINE authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

IMPLICIT_SCHEMA
Revokes the authority to implicitly create a schema. It does not affect the ability to create objects in existing schemas or to process a CREATE SCHEMA statement.

DBADM
Revokes the DBADM authority.

DBADM authority cannot be revoked from PUBLIC (because it cannot be granted to PUBLIC).

CAUTION:
Revoking DBADM authority does not automatically revoke any privileges that were held by the *authorization-name* **on objects in the database, nor does it revoke any of the other database authorities that were implicitly and automatically granted when DBADM authority was originally granted.**

LOAD
Revokes the authority to LOAD in this database.

QUIESCE_CONNECT
Revokes the authority to access the database while it is quiesced.

FROM
Indicates from whom the authorities are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used (SQLSTATE 42502). It is not possible to revoke the authorities from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.

PUBLIC
Revokes the authorities from PUBLIC.

BY ALL
Revokes each named privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

Rules:

- If neither USER nor GROUP is specified, then:

- If all rows for the grantee in the SYSCAT.DBAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
- If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
- If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
- If DCE authentication is used, then an error is raised (SQLSTATE 56092).

Notes:

- ***Compatibilities***

  For compatibility with versions earlier than Version 8, the option CREATE_NOT_FENCED can be substituted for CREATE_NOT_FENCED_ROUTINE.
- Revoking a specific privilege does not necessarily revoke the ability to perform an action. A user may proceed with a task if other privileges are held by PUBLIC or a group, or if the user has a higher level authority, such as DBADM.

Examples:

Example 1: Given that USER6 is only a user and not a group, revoke the privilege to create tables from the user USER6.

```
REVOKE CREATETAB ON DATABASE FROM USER6
```

Example 2: Revoke BINDADD authority on the database from a group named D024. There are two rows in the SYSCAT.DBAUTH catalog view for this grantee; one with a GRANTEETYPE of U and one with a GRANTEETYPE of G.

```
REVOKE BINDADD ON DATABASE FROM GROUP D024
```

In this case, the GROUP keyword must be specified; otherwise an error will occur (SQLSTATE 56092).

Related reference:

- "REVOKE (Index Privileges)" on page 647
- "REVOKE (Package Privileges)" on page 650
- "REVOKE (Schema Privileges)" on page 657
- "REVOKE (Table, View, or Nickname Privileges)" on page 662
- "REVOKE (Server Privileges)" on page 660
- "REVOKE (Table Space Privileges)" on page 668
- "REVOKE (Routine Privileges)" on page 653

Related samples:

- “dbauth.sqc -- How to grant, display, and revoke authorities at database level (C)”
- “dbauth.sqC -- How to grant, display, and revoke authorities at database level (C++)”
- “DbAuth.java -- Grant, display or revoke privileges on database (JDBC)”
- “DbAuth.sqlj -- Grant, display or revoke privileges on database (SQLj)”

REVOKE (Index Privileges)

This form of the REVOKE statement revokes the CONTROL privilege on an index.

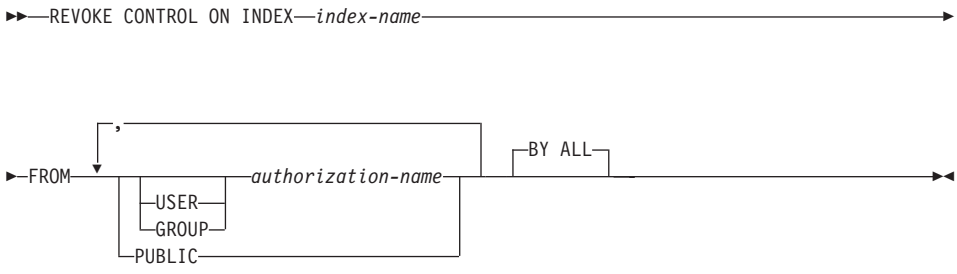
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement must hold either SYSADM or DBADM authority (SQLSTATE 42501).

Syntax:



Description:

CONTROL
Revokes the privilege to drop the index. This is the CONTROL privilege for indexes, which is automatically granted to creators of indexes.

ON INDEX *index-name*
Specifies the name of the index on which the CONTROL privilege is to be revoked.

FROM
Indicates from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used (SQLSTATE 42502). It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.

PUBLIC
Revokes the privileges from PUBLIC.

BY ALL
Revokes the privilege from all named users who were explicitly granted that privilege, regardless of who granted it. This is the default behavior.

Rules:

- If neither USER nor GROUP is specified, then:
  If all rows for the grantee in the SYSCAT.INDEXAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  If DCE authentication is used, then an error is raised (SQLSTATE 56092).

Notes:

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have authorities such as ALTERIN on the schema of an index.

Examples:

Example 1: Given that USER4 is only a user and not a group, revoke the privilege to drop an index DEPTIDX from the user USER4.

```
REVOKE CONTROL ON INDEX DEPTIDX FROM USER4
```

Example 2: Revoke the privilege to drop an index LUNCHITEMS from the user CHEF and the group WAITERS.

```
REVOKE CONTROL ON INDEX LUNCHITEMS
  FROM USER CHEF, GROUP WAITERS
```

Related reference:

- "REVOKE (Database Authorities)" on page 642
- "REVOKE (Package Privileges)" on page 650
- "REVOKE (Schema Privileges)" on page 657
- "REVOKE (Table, View, or Nickname Privileges)" on page 662
- "REVOKE (Server Privileges)" on page 660
- "REVOKE (Table Space Privileges)" on page 668

- "REVOKE (Routine Privileges)" on page 653

REVOKE (Package Privileges)

This form of the REVOKE statement revokes CONTROL, BIND, and EXECUTE privileges against a package.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

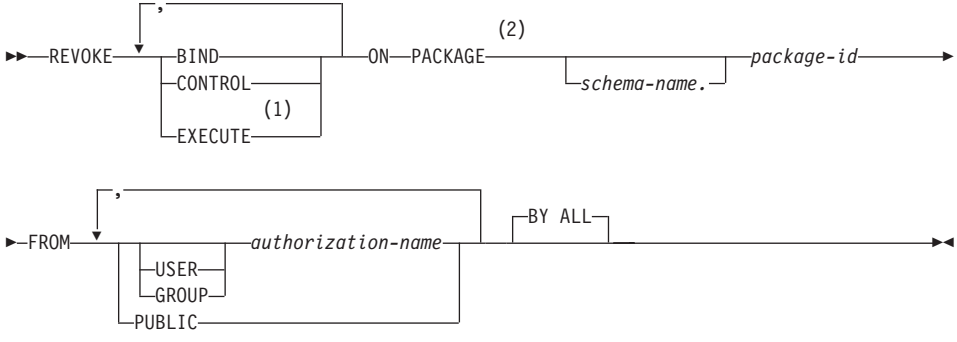
Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- CONTROL privilege on the referenced package
- SYSADM or DBADM authority.

To revoke the CONTROL privilege, SYSADM or DBADM authority are required.

Syntax:



Notes:

1 RUN can be used as a synonym for EXECUTE.

2 PROGRAM can be used as a synonym for PACKAGE.

Description:

BIND
Revokes the privilege to execute BIND or REBIND on—or to add a new version of— the referenced package.

The BIND privilege cannot be revoked from an *authorization-name* that holds CONTROL privilege on the package, without also revoking the CONTROL privilege.

CONTROL
Revokes the privilege to drop the package and to extend package privileges to other users.

Revoking CONTROL does not revoke the other package privileges.

EXECUTE
Revokes the privilege to execute the package.

The EXECUTE privilege cannot be revoked from an *authorization-name* that holds CONTROL privilege on the package without also revoking the CONTROL privilege.

ON PACKAGE *schema-name.package-id*
Specifies the name of the package on which privileges are to be revoked. If a schema name is not specified, the package ID is implicitly qualified by the default schema. The revoking of a package privilege applies to all versions of the package.

FROM
Indicates from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used (SQLSTATE 42502). It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.

PUBLIC
Revokes the privileges from PUBLIC.

BY ALL
Revokes each named privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

Rules:

- If neither USER nor GROUP is specified, then:

- If all rows for the grantee in the SYSCAT.PACKAGEAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
- If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
- If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
- If DCE authentication is used, then an error is raised (SQLSTATE 56092).

Notes:

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have privileges such as ALTERIN on the schema of a package.

Examples:

Example 1: Revoke the EXECUTE privilege on package CORPDATA.PKGA from PUBLIC.

```
REVOKE EXECUTE
  ON PACKAGE CORPDATA.PKGA
  FROM PUBLIC
```

Example 2: Revoke CONTROL authority on the RRSP_PKG package for the user FRANK and for PUBLIC.

```
REVOKE CONTROL
  ON PACKAGE RRSP_PKG
  FROM USER FRANK, PUBLIC
```

Related reference:

- "REVOKE (Database Authorities)" on page 642
- "REVOKE (Index Privileges)" on page 647
- "REVOKE (Schema Privileges)" on page 657
- "REVOKE (Table, View, or Nickname Privileges)" on page 662
- "REVOKE (Server Privileges)" on page 660
- "REVOKE (Table Space Privileges)" on page 668
- "REVOKE (Routine Privileges)" on page 653

REVOKE (Routine Privileges)

This form of the REVOKE statement revokes privileges on a routine (function, method, or procedure).

Invocation:

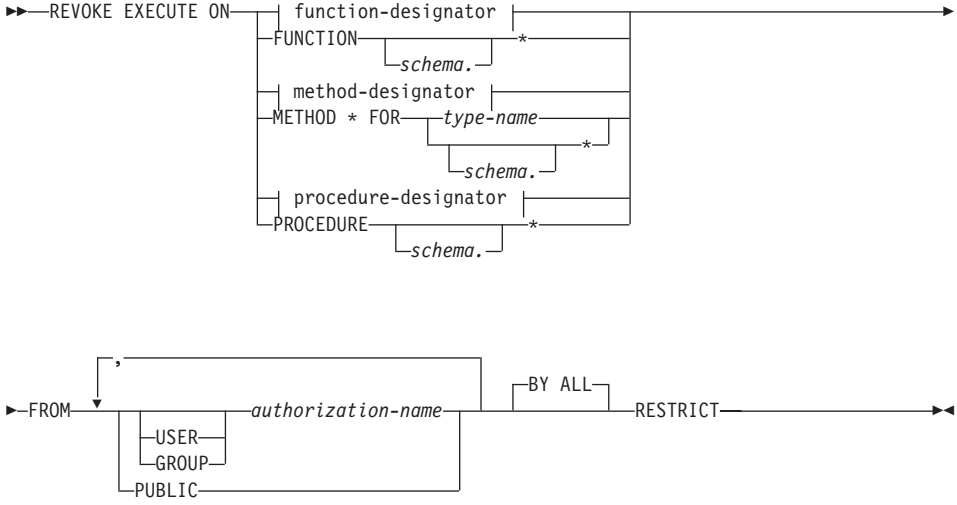
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority

Syntax:



Description:

EXECUTE
Revokes the privilege to run the identified user-defined function, method, or stored procedure.

function-designator
Uniquely identifies the function.

FUNCTION *schema.**
Identifies the explicit grant for all the existing and future functions in the schema. Revoking the *schema*.* privilege does not revoke any privileges that were granted on a specific function. In dynamic SQL statements, if a schema is not specified, the schema in the CURRENT SCHEMA special register will be used. In static SQL statements, if a schema is not specified, the schema in the QUALIFIER precompile/bind option will be used.

method-designator
Uniquely identifies the method.

METHOD *
Identifies the explicit grant for all the existing and future methods for the type *type-name*. Revoking the * privilege does not revoke any privileges that were granted on a specific method.

FOR *type-name*
Names the type in which the specified method is found. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the value of the CURRENT SCHEMA special register is used as a qualifier for an unqualified type name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified type names. An asterisk (*) can be used in place of *type-name* to identify the explicit grant on all existing and future methods for all existing and future types in the schema. Revoking the privilege using an asterisk for method and *type-name* does not revoke any privileges that were granted on a specific method or on all methods for a specific type.

procedure-designator
Uniquely identifies the procedure.

PROCEDURE *schema.**
Identifies the explicit grant for all the existing and future procedures in the schema. Revoking the *schema*.* privilege does not revoke any privileges that were granted on a specific procedure. In dynamic SQL statements, if a schema is not specified, the schema in the CURRENT SCHEMA special register will be used. In static SQL statements, if a schema is not specified, the schema in the QUALIFIER precompile/bind option will be used.

FROM
Specifies from whom the EXECUTE privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups. The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the EXECUTE privilege from all users.

BY ALL
Revokes the EXECUTE privilege from all named users who were explicitly granted the privilege, regardless of who granted it. This is the default behavior.

RESTRICT
Specifies that the EXECUTE privilege cannot be revoked if both of the following are true (SQLSTATE 42893):

- The specified routine is used in a view, trigger, constraint, index extension, SQL function, SQL method, transform group, or is referenced as the SOURCE of a sourced function.
- The loss of the EXECUTE privilege would cause the definer of the view, trigger, constraint, index extension, SQL function, SQL method, transform group, or sourced function to no longer be able to execute the specified routine.

Rules:

- It is not possible to revoke the EXECUTE privilege on a function or method defined with schema 'SYSIBM' or 'SYSFUN' (SQLSTATE 42832).
- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.ROUTINEAUTH catalog views have a GRANTEETYPE of U, then USER is assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP is assumed.
  - If some rows have U and some rows have G, an error (SQLSTATE 56092) is raised.
  - If DCE authentication is used, an error is raised (SQLSTATE 56092).

Examples:

Example 1: Revoke the EXECUTE privilege on function CALC_SALARY from user JONES. Assume that there is only one function in the schema with function name CALC_SALARY.

```
REVOKE EXECUTE ON FUNCTION CALC_SALARY FROM JONES RESTRICT
```

Example 2: Revoke the EXECUTE privilege on procedure VACATION_ACCR from all users at the current server.

```
REVOKE EXECUTE ON PROCEDURE VACATION_ACCR FROM PUBLIC RESTRICT
```

Example 3: Revoke the EXECUTE privilege on function NEW_DEPT_HIRES from HR (Human Resources). The function has two input parameters of type INTEGER and CHAR(10), respectively. Assume that the schema has more than one function named NEW_DEPT_HIRES.

```
REVOKE EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10))
  FROM HR RESTRICT
```

Example 4: Revoke the EXECUTE privilege on method SET_SALARY for type EMPLOYEE from user Jones.

```
REVOKE EXECUTE ON METHOD SET_SALARY FOR EMPLOYEE FROM JONES RESTRICT
```

Related reference:

- "REVOKE (Database Authorities)" on page 642
- "REVOKE (Index Privileges)" on page 647
- "REVOKE (Package Privileges)" on page 650
- "REVOKE (Schema Privileges)" on page 657
- "REVOKE (Table, View, or Nickname Privileges)" on page 662
- "REVOKE (Server Privileges)" on page 660
- "REVOKE (Table Space Privileges)" on page 668
- "Common syntax elements" on page xi

REVOKE (Schema Privileges)

This form of the REVOKE statement revokes the privileges on a schema.

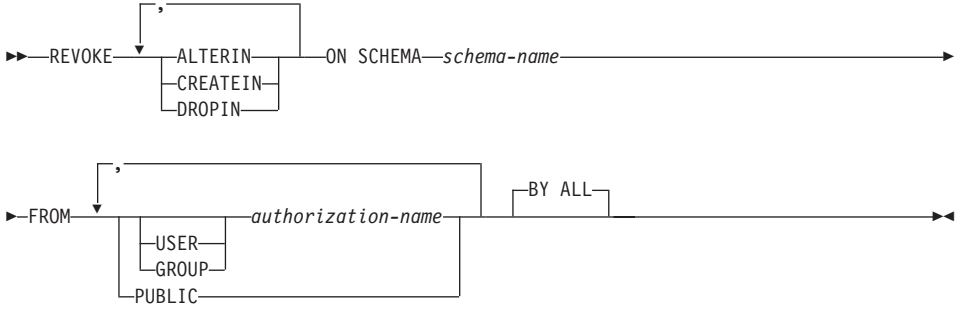
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement must hold either SYSADM or DBADM authority (SQLSTATE 42501).

Syntax:



Description:

ALTERIN
Revokes the privilege to alter or comment on objects in the schema.

CREATEIN
Revokes the privilege to create objects in the schema.

DROPIN
Revokes the privilege to drop objects in the schema.

ON SCHEMA *schema-name*
Specifies the name of the schema on which privileges are to be revoked.

FROM
Indicates from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used (SQLSTATE 42502). It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.

PUBLIC
Revokes the privileges from PUBLIC.

BY ALL
Revokes each named privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

Rules:

- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.SCHEMAAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  - If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  - If DCE authentication is used, then an error is raised (SQLSTATE 56092).

Notes:

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have a higher level authority such as DBADM.

Examples:

Example 1: Given that USER4 is only a user and not a group, revoke the privilege to create objects in schema DEPTIDX from the user USER4.

```
REVOKE CREATEIN ON SCHEMA DEPTIDX FROM USER4
```

Example 2: Revoke the privilege to drop objects in schema LUNCH from the user CHEF and the group WAITERS.

```
REVOKE DROPIN ON SCHEMA LUNCH
  FROM USER CHEF, GROUP WAITERS
```

Related reference:

- "REVOKE (Database Authorities)" on page 642
- "REVOKE (Index Privileges)" on page 647
- "REVOKE (Package Privileges)" on page 650
- "REVOKE (Table, View, or Nickname Privileges)" on page 662
- "REVOKE (Server Privileges)" on page 660
- "REVOKE (Table Space Privileges)" on page 668
- "REVOKE (Routine Privileges)" on page 653

REVOKE (Server Privileges)

This form of the REVOKE statement revokes the privilege to access and use a specified data source in pass-through mode.

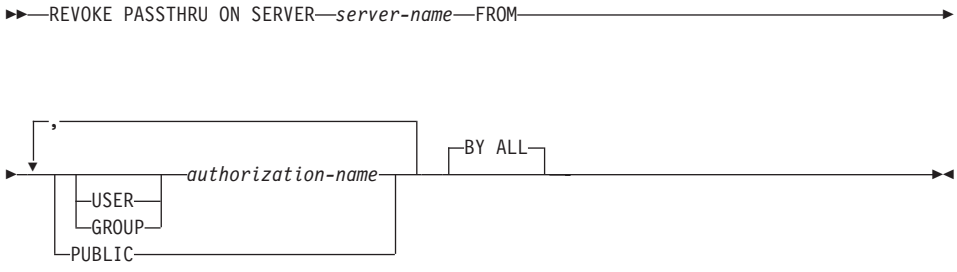
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement must have SYSADM or DBADM authority.

Syntax:



Description:

SERVER *server-name*
Names the data source for which the privilege to use in pass-through mode is being revoked. *server-name* must identify a data source that is described in the catalog.

FROM
Specifies from whom the privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

The authorization ID of the REVOKE statement itself cannot be used (SQLSTATE 42502). It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.

PUBLIC
Revokes from all users the privilege to pass through to *server-name*.

BY ALL
Revokes the privilege from all named users who were explicitly granted that privilege, regardless of who granted it. This is the default behavior.

Examples:

Example 1: Revoke USER6's privilege to pass through to data source MOUNTAIN.

```
REVOKE PASSTHRU ON SERVER MOUNTAIN FROM USER USER6
```

Example 2: Revoke group D024's privilege to pass through to data source EASTWING.

```
REVOKE PASSTHRU ON SERVER EASTWING FROM GROUP D024
```

The members of group D024 will no longer be able to use their group ID to pass through to EASTWING. But if any members have the privilege to pass through to EASTWING under their own user IDs, they will retain this privilege.

Related reference:

- "REVOKE (Database Authorities)" on page 642
- "REVOKE (Index Privileges)" on page 647
- "REVOKE (Package Privileges)" on page 650
- "REVOKE (Schema Privileges)" on page 657
- "REVOKE (Table, View, or Nickname Privileges)" on page 662
- "REVOKE (Table Space Privileges)" on page 668
- "REVOKE (Routine Privileges)" on page 653

REVOKE (Table, View, or Nickname Privileges)

This form of the REVOKE statement revokes privileges on a table, view, or nickname.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

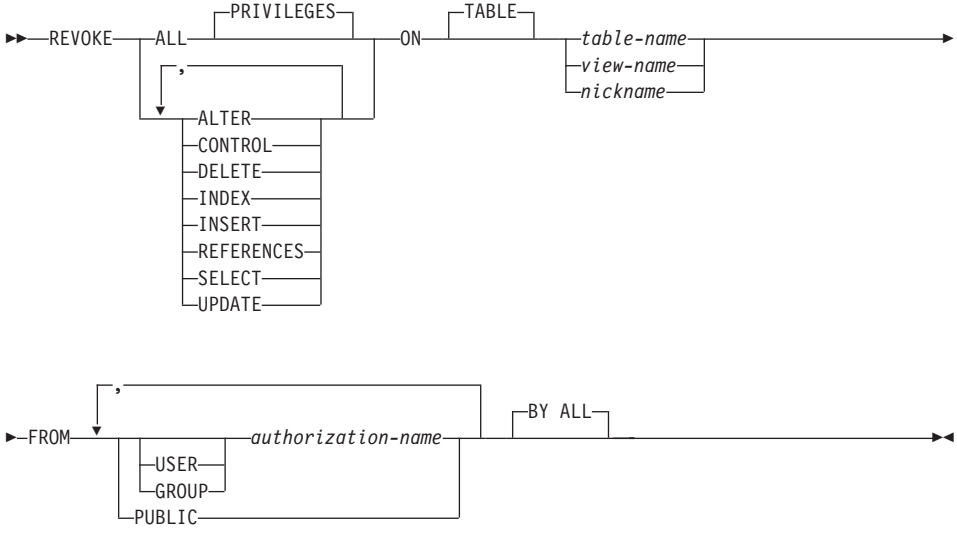
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CONTROL privilege on the referenced table, view, or nickname.

To revoke the CONTROL privilege, either SYSADM or DBADM authority is required.

To revoke the privileges on catalog tables and views, either SYSADM or DBADM authority is required.

Syntax:



Description:

ALL or **ALL PRIVILEGES**
Revokes all privileges (except CONTROL) held by an authorization-name for the specified tables, views, or nicknames.

If ALL is not used, one or more of the keywords listed below must be used. Each keyword revokes the privilege described, but only as it applies to the tables, views, or nicknames named in the ON clause. The same keyword must not be specified more than once.

ALTER
Revokes the privilege to add columns to the base table definition; create or drop a primary key or unique constraint on the table; create or drop a foreign key on the table; add/change a comment on the table, view, or nickname; create or drop a check constraint; create a trigger; add, reset, or drop a column option for a nickname; or, change nickname column names or data types.

CONTROL
Revokes the ability to drop the table, view, or nickname, and the ability to execute the RUNSTATS utility on the table and indexes.

Revoking CONTROL privilege from an *authorization-name* does not revoke other privileges granted to the user on that object.

DELETE
Revokes the privilege to delete rows from the table, updatable view, or nickname.

INDEX
Revokes the privilege to create an index on the table or an index specification on the nickname. The creator of an index or index specification automatically has the CONTROL privilege over the index or index specification (authorizing the creator to drop the index or index specification). In addition, the creator retains this privilege even if the INDEX privilege is revoked.

INSERT
Revokes the privileges to insert rows into the table, updatable view, or nickname, and to run the IMPORT utility.

REFERENCES
Revokes the privilege to create or drop a foreign key referencing the table as the parent. Any column level REFERENCES privileges are also revoked.

SELECT
Revokes the privilege to retrieve rows from the table or view, to create a view on a table, and to run the EXPORT utility against the table or view.

Revoking SELECT privilege may cause some views to be marked inoperative. (For information on inoperative views, see “CREATE VIEW”.)

UPDATE
Revokes the privilege to update rows in the table, updatable view, or nickname. Any column level UPDATE privileges are also revoked.

ON TABLE *table-name* or *view-name* or *nickname*
Specifies the table, view, or nickname on which privileges are to be revoked. The *table-name* cannot be a declared temporary table (SQLSTATE 42995).

FROM
Indicates from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists one or more authorization IDs.

The ID of the REVOKE statement itself cannot be used (SQLSTATE 42502). It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.

PUBLIC
Revokes the privileges from PUBLIC.

BY ALL
Revokes each named privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

Rules:

- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.TABAUTH and SYSCAT.COLAUTH catalog views have a GRANTEETYPE of U, then USER will be assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  - If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  - If DCE authentication is used, then an error is raised (SQLSTATE 56092).

Notes:

- If a privilege is revoked from the *authorization-name* used to create a view (this is called the view's DEFINER in SYSCAT.VIEWS), that privilege is also revoked from any dependent views.

- If the DEFINER of the view loses a SELECT privilege on some object on which the view definition depends (or an object upon which the view definition depends is dropped, or made inoperative in the case of another view), the view will be made inoperative.

  However, if a DBADM or SYSADM explicitly revokes all privileges on the view from the DEFINER, then the record of the DEFINER will not appear in SYSCAT.TABAUTH but nothing will happen to the view - it remains operative.
- Privileges on inoperative views cannot be revoked.
- All packages dependent upon an object for which a privilege is revoked are marked invalid. A package remains invalid until a bind or rebind operation on the application is successfully executed, or the application is executed and the database manager successfully rebinds the application (using information stored in the catalogs). Packages marked invalid due to a revoke may be successfully rebound without any additional grants.

  For example, if a package owned by USER1 contains a SELECT from table T1 and the SELECT privilege for table T1 is revoked from USER1, then the package will be marked invalid. If SELECT authority is re-granted, or if the user holds DBADM authority, the package is successfully rebound when executed.
- Packages, triggers or views that include the use of OUTER(*Z*) in the FROM clause, are dependent on having SELECT privilege on every subtable or subview of *Z*. Similarly, packages, triggers, or views that include the use of DEREF(*Y*) where *Y* is a reference type with a target table or view *Z*, are dependent on having SELECT privilege on every subtable or subview of *Z*. If one of these SELECT privileges is revoked, such packages are invalidated and such triggers or views are made inoperative.
- Table, view, or nickname privileges cannot be revoked from an *authorization-name* with CONTROL on the object without also revoking the CONTROL privilege (SQLSTATE 42504).
- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have privileges such as ALTERIN on the schema of a table or a view.
- If the DEFINER of the materialized query table loses a SELECT privilege on a table on which the materialized query table definition depends (or a table upon which the materialized query table definition depends is dropped), the materialized query table will be made inoperative.

  However, if a DBADM or SYSADM explicitly revokes all privileges on the materialized query table from the DEFINER, then the record in SYSTABAUTH for the DEFINER will be deleted, but nothing will happen to the materialized query table - it remains operative.

- Revoking nickname privileges has no affect on data source object (table or view) privileges.
- Revoking the SELECT privilege for a table or view that is directly or indirectly referenced in an SQL function or method body may fail if the SQL function or method body cannot be dropped because some other object is dependent on it (SQLSTATE 42893).
- If the DEFINER of the SQL function or method body loses the SELECT privilege on some object on which the function or method body definition depends (or if an object upon which the function or method body definition depends is dropped), the function or method body will be dropped, unless another object depends on the function or method (SQLSTATE 42893).

Examples:

Example 1: Revoke SELECT privilege on table EMPLOYEE from user ENGLES. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is U.

```
REVOKE SELECT
  ON TABLE EMPLOYEE
  FROM ENGLES
```

Example 2: Revoke update privileges on table EMPLOYEE previously granted to all local users. Note that grants to specific users are not affected.

```
REVOKE UPDATE
  ON EMPLOYEE
  FROM PUBLIC
```

Example 3: Revoke all privileges on table EMPLOYEE from users PELLOW and MLI and from group PLANNERS.

```
REVOKE ALL
  ON EMPLOYEE
  FROM USER PELLOW, USER MLI, GROUP PLANNERS
```

Example 4: Revoke SELECT privilege on table CORPDATA.EMPLOYEE from a user named JOHN. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is U.

```
REVOKE SELECT
  ON CORPDATA.EMPLOYEE FROM JOHN
```

or

```
REVOKE SELECT
  ON CORPDATA.EMPLOYEE FROM USER JOHN
```

Note that an attempt to revoke the privilege from GROUP JOHN would result in an error, since the privilege was not previously granted to GROUP JOHN.

Example 5: Revoke SELECT privilege on table CORPDATA.EMPLOYEE from a group named JOHN. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is G.

```
REVOKE SELECT
  ON CORPDATA.EMPLOYEE FROM JOHN
```

or

```
REVOKE SELECT
  ON CORPDATA.EMPLOYEE FROM GROUP JOHN
```

Example 6: Revoke user SHAWN's privilege to create an index specification on nickname ORAREM1.

```
REVOKE INDEX
  ON ORAREM1 FROM USER SHAWN
```

Related reference:

- "CREATE TABLE" on page 332
- "CREATE VIEW" on page 463
- "DROP" on page 512
- "REVOKE (Database Authorities)" on page 642
- "REVOKE (Index Privileges)" on page 647
- "REVOKE (Package Privileges)" on page 650
- "REVOKE (Schema Privileges)" on page 657
- "REVOKE (Server Privileges)" on page 660
- "REVOKE (Table Space Privileges)" on page 668
- "REVOKE (Routine Privileges)" on page 653

Related samples:

- "tbpriv.sqc -- How to grant, display, and revoke privileges (C)"
- "tbpriv.sqC -- How to grant, display, and revoke privileges (C++)"
- "TbPriv.java -- How to grant, display and revoke privileges on a table (JDBC)"
- "TbPriv.sqlj -- How to grant, display and revoke privileges on a table (SQLj)"

REVOKE (Table Space Privileges)

This form of the REVOKE statement revokes the USE privilege on a table space.

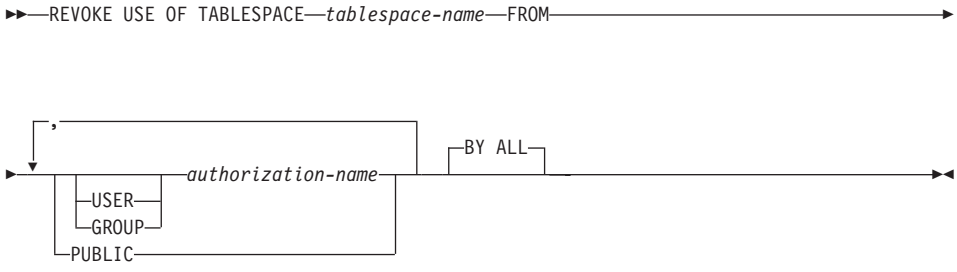
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement must hold either SYSADM, SYSCTRL or DBADM authority (SQLSTATE 42501).

Syntax:



Description:

USE
Revokes the privilege to specify or default to the table space when creating a table.

OF TABLESPACE *tablespace-name*
Specifies the table space on which the USE privilege is to be revoked. The table space cannot be SYSCATSPACE (SQLSTATE 42838) or a SYSTEM TEMPORARY table space (SQLSTATE 42809).

FROM
Indicates from whom the USE privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name
Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used (SQLSTATE 42502). It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.

PUBLIC
Revokes the USE privilege from PUBLIC.

BY ALL
Revokes the privilege from all named users who were explicitly granted that privilege, regardless of who granted it. This is the default behavior.

Rules:

- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.TBSPACEAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  - If some rows have U and some rows have G, then an error results (SQLSTATE 56092).
  - If DCE authentication is used, then an error results (SQLSTATE 56092).

Notes:

- Revoking the USE privilege does not necessarily revoke the ability to create tables in that table space. A user may still be able to create tables in that table space if the USE privilege is held by PUBLIC or a group, or if the user has a higher level authority, such as DBADM.

Examples:

Example 1: Revoke the privilege to create tables in table space PLANS from the user BOBBY.

```
REVOKE USE OF TABLESPACE PLANS FROM USER BOBBY
```

Related reference:

- "REVOKE (Database Authorities)" on page 642
- "REVOKE (Index Privileges)" on page 647
- "REVOKE (Package Privileges)" on page 650
- "REVOKE (Schema Privileges)" on page 657
- "REVOKE (Table, View, or Nickname Privileges)" on page 662
- "REVOKE (Server Privileges)" on page 660

- "REVOKE (Routine Privileges)" on page 653

ROLLBACK

The ROLLBACK statement is used to back out of the database changes that were made within a unit of work or a savepoint.

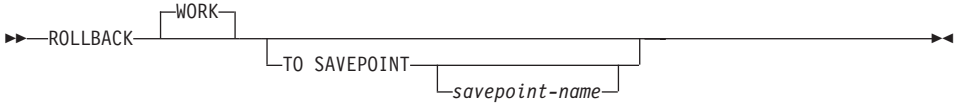
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

None required.

Syntax:



Description:
The unit of work in which the ROLLBACK statement is executed is terminated and a new unit of work is initiated. All changes made to the database during the unit of work are backed out.

The following statements, however, are not under transaction control, and changes made by them are independent of the ROLLBACK statement:

- SET CONNECTION
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT PACKAGESET
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE
- SET PASSTHRU

Note: Although the SET PASSTHRU statement is not under transaction control, the passthru session initiated by the statement is under transaction control.

- SET PATH
- SET SCHEMA
- SET SERVER OPTION

The generation of sequence and identity values is not under transaction control. Values generated and consumed by the nextval-expression or by inserting rows into a table that has an identity column are independent of issuing the ROLLBACK statement. Also, issuing the ROLLBACK statement does not affect the value returned by the prevval-expression, nor the IDENTITY_VAL_LOCAL function.

TO SAVEPOINT
Indicates that a partial rollback (ROLLBACK TO SAVEPOINT) is to be performed. If no savepoint is active, an SQL error is returned (SQLSTATE 3B502). After a successful ROLLBACK, the savepoint continues to exist. If a *savepoint-name* is not provided, rollback is to the most recently set savepoint.

If this clause is omitted, the ROLLBACK WORK statement rolls back the entire transaction. Furthermore, savepoints within the transaction are released.

savepoint-name
Indicate the savepoint to which to rollback. After a successful ROLLBACK, the savepoint defined by *savepoint-name* continues to exist. If the savepoint name does not exist, an error is returned (SQLSTATE 3B001). Data and schema changes made since the savepoint was set are undone.

Notes:

- All locks held are released on a ROLLBACK of the unit of work. All open cursors are closed. All LOB locators are freed.
- Executing a ROLLBACK statement does not affect either the SET statements that change special register values or the RELEASE statement.
- If the program terminates abnormally, the unit of work is implicitly rolled back.
- Statement caching is affected by the rollback operation.
- Savepoints are not allowed in atomic execution contexts such as atomic compound statements and triggers.
- The impact on cursors resulting from a ROLLBACK TO SAVEPOINT depends on the statements within the savepoint

  - If the savepoint contains DDL on which a cursor is dependent, the cursor is marked invalid. Attempts to use such a cursor results in an error (SQLSTATE 57007).
  - Otherwise:
    - If the cursor is referenced in the savepoint, the cursor remains open and is positioned before the next logical row of the result table. (A FETCH must be performed before a positioned UPDATE or DELETE statement is issued.)
    - Otherwise, the cursor is not affected by the ROLLBACK TO SAVEPOINT (it remains open and positioned).
- Dynamically prepared statement names are still valid, although the statement may be implicitly prepared again, as a result of DDL operations that are rolled back within the savepoint.
- A ROLLBACK TO SAVEPOINT operation will drop any declared temporary tables named within the savepoint. If a declared temporary table is modified within the savepoint, then all rows in the table are deleted.
- All locks are retained after a ROLLBACK TO SAVEPOINT statement.
- All LOB locators are preserved following a ROLLBACK TO SAVEPOINT operation.

Example:

Delete the alterations made since the last commit point or rollback.

```
ROLLBACK WORK
```

Related reference:

- "EXECUTE" on page 544

Related samples:

- "delet.sqb -- How to delete table data (MF COBOL)"
- "spclient.sqc -- Call various stored procedures (C)"
- "tut_use.sqc -- How to modify a database (C)"
- "spclient.sqC -- Call various stored procedures (C++)"
- "tut_use.sqC -- How to modify a database (C++)"

SAVEPOINT

Use the SAVEPOINT statement to set a savepoint within a transaction.

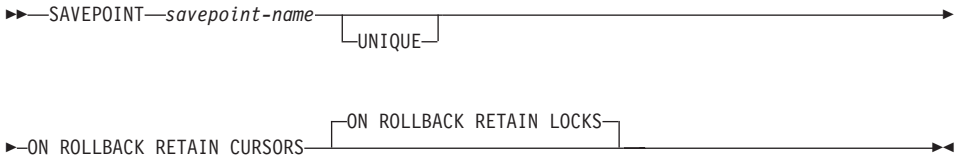
Invocation:

This statement can be imbedded in an application program (including a stored procedure) or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization:

None required.

Syntax:



Description:

savepoint-name
Name of the *savepoint*. The specified *savepoint-name* cannot begin with 'SYS' (SQLSTATE 42939).

UNIQUE
Specifying a UNIQUE savepoint indicates that the application does not intend to reuse this savepoint name while the savepoint is active.

ON ROLLBACK RETAIN CURSORS
Specifies system behavior upon rollback to this savepoint with respect to open cursor statements processed after the SAVEPOINT statement. The RETAIN CURSORS clause indicates that, whenever possible, the cursors are unchanged by a rollback to savepoint. For situations where the cursors are affected by the rollback to savepoint, see "ROLLBACK".

ON ROLLBACK RETAIN LOCKS
Specifies system behavior upon rollback to this savepoint with respect to locks acquired after the setting of the savepoint. Locks acquired since the savepoint are not tracked and are not rolled back (released) on rollback to the savepoint.

Rules:

- Savepoints cannot be nested. If a savepoint statement is issued, and there is already an established savepoint present, then an error occurs (SQLSTATE 3B002).

Notes:

- Once a SAVEPOINT statement has been issued, insert, update, or delete operations on nicknames are not allowed.
- The UNIQUE keyword is supported for compatibility with DB2 Universal Database for OS/390 and z/OS. The following describes the behavior on DB2 Universal Database for OS/390 and z/OS.

  If a savepoint named *savepoint-name* already exists within the transaction, an error is returned (SQLSTATE 3B501). By omitting the UNIQUE clause, the applications assert that this savepoint name may be reused within the transaction. If *savepoint-name* already exists within the transaction, it will be destroyed and a new savepoint named *savepoint-name* will be created.

  Destruction of a savepoint by reusing its name for another savepoint is not the same as releasing the old savepoint with the RELEASE SAVEPOINT statement. Destruction of a savepoint by reusing its name destroys just that savepoint. Releasing a savepoint by means of the RELEASE SAVEPOINT statement releases the named savepoint and all savepoints established after the named savepoint.
- Within a savepoint, if a utility, SQL statement, or DB2 command performs intermittent COMMIT statements during processing, then the savepoint will be implicitly released.
- The SQL statement SET INTEGRITY has the same effects as a DDL statement within a savepoint.
- In an application, inserts may be buffered (that is, the application was precompiled with INSERT BUF option). The buffer will be flushed when SAVEPOINT, ROLLBACK, or RELEASE TO SAVEPOINT statements are issued.

Related reference:

- "ROLLBACK" on page 671

SELECT

The SELECT statement is a form of query. It can be embedded in an application program or issued interactively.

Related reference:

- "Subselect" in the *SQL Reference, Volume 1*
- "Select-statement" in the *SQL Reference, Volume 1*

Related samples:

- "dynamic.sqb -- How to update table data with cursor dynamically (MF COBOL)"
- "static.sqb -- Get table data using static SQL statement (MF COBOL)"
- "tbread.c -- How to read data from tables (CLI)"
- "tut_read.c -- How to read data from tables (CLI)"
- "tbread.sqc -- How to read tables (C)"
- "tut_read.sqc -- How to read tables (C)"
- "tbread.sqC -- How to read tables (C++)"
- "tut_read.sqC -- How to read tables (C++)"
- "TbRead.java -- How to read table data (JDBC)"
- "TutRead.java -- Read data in a table (JDBC)"
- "TbRead.sqlj -- How to read table data (SQLj)"
- "TutRead.sqlj -- Read data in a table (SQLj)"

SELECT INTO

The SELECT INTO statement produces a result table consisting of at most one row, and assigns the values in that row to host variables. If the table is empty, the statement assigns +100 to SQLCODE and '02000' to SQLSTATE and does not assign values to the host variables. If more than one row satisfies the search condition, statement processing is terminated, and an error occurs (SQLSTATE 21000).

Invocation:

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization:

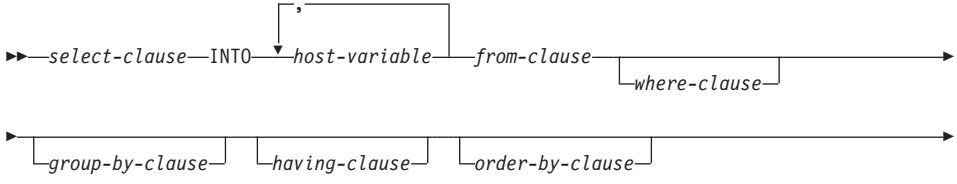
The privileges held by the authorization ID of the statement must include at least one of the following:

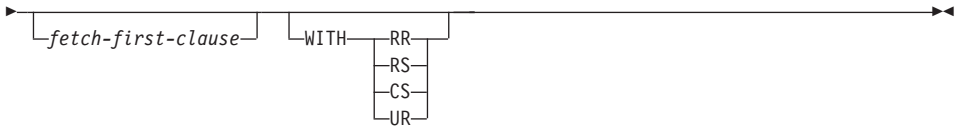
- SELECT privilege on the table, view, or nickname
- CONTROL privilege on the table, view, or nickname
- SYSADM or DBADM authority

GROUP privileges are not checked for static SELECT INTO statements.

If the target of the SELECT INTO statement is a nickname, the privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges required for the operation on the object at the data source. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

Syntax:





Description:
For a description of the *select-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*, and *fetch-first-clause*, see "Queries" in the *SQL Reference, Volume 2*.

INTO
Introduces a list of host variables.

host-variable
Identifies a variable that is described in the program under the rules for declaring host 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. If the number of host variables is less than the number of column values, the value 'W' is assigned to the SQLWARN3 field of the SQLCA.

Each assignment to a variable is made in sequence through the list. If an error occurs, no value is assigned to any host variable.

WITH
Specifies the isolation level at which the SELECT INTO statement is executed.

RR
Repeatable Read

RS
Read Stability

CS
Cursor Stability

UR
Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound.

Examples:

Example 1: This C example puts the maximum salary in EMP into the host variable MAXSALARY.

```
EXEC SQL SELECT MAX(SALARY)
  INTO :MAXSALARY
  FROM EMP;
```

Example 2: This C example puts the row for employee 528671, from EMP, into host variables.

```
EXEC SQL SELECT * INTO :h1, :h2, :h3, :h4
  FROM EMP
  WHERE EMPNO = '528671';
```

Related concepts:

- "Queries" in the *SQL Reference, Volume 1*

Related reference:

- "SQLCA (SQL communications area)" in the *SQL Reference, Volume 1*
- "Assignments and comparisons" in the *SQL Reference, Volume 1*

Related samples:

- "dbauth.sqc -- How to grant, display, and revoke authorities at database level (C)"
- "dtlob.sqc -- How to use the LOB data type (C)"
- "spclient.sqc -- Call various stored procedures (C)"
- "spserver.sqc -- A variety of types of stored procedures (C)"
- "tbreorg.sqc -- How to reorganize a table and update its statistics (C)"
- "dbauth.sqC -- How to grant, display, and revoke authorities at database level (C++)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "spclient.sqC -- Call various stored procedures (C++)"
- "spserver.sqC -- A variety of types of stored procedures (C++)"
- "tbreorg.sqC -- How to reorganize a table and update its statistics (C++)"

SET CONNECTION

The SET CONNECTION statement changes the state of a connection from dormant to current, making the specified location the current server. It is not under transaction control.

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:

None Required.

Syntax:

Description:

server-name or *host-variable*
Identifies the application server by the specified *server-name* or a *host-variable* which contains the *server-name*.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-justified and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The *server-name* or the *host-variable* must identify an existing connection of the application process. If they do not identify an existing connection, an error (SQLSTATE 08003) is raised.

If SET CONNECTION is to the current connection, the states of all connections of the application process are unchanged.

Successful Connection

If the SET CONNECTION statement executes successfully:

- No connection is made. The CURRENT SERVER special register is updated with the specified *server-name*.

- The previously current connection, if any, is placed into the dormant state (assuming a different *server-name* is specified).
- The CURRENT SERVER special register and the SQLCA are updated in the same way as documented under "CONNECT (Type 1)".

Unsuccessful Connection

If the SET CONNECTION statement fails:

- No matter what the reason for failure, the connection state of the application process and the states of its connections are unchanged.
- As with an unsuccessful Type 1 CONNECT, the SQLERRP field of the SQLCA is set to the name of the module that detected the error.

Notes:

- The use of type 1 CONNECT statements does not preclude the use of SET CONNECTION, but the statement will always fail (SQLSTATE 08003), unless the SET CONNECTION statement specifies the current connection, because dormant connections cannot exist.
- The SQLRULES(DB2) connection option (see "Options that Govern Distributed Unit of Work Semantics") does not preclude the use of SET CONNECTION, but the statement is unnecessary, because type 2 CONNECT statements can be used instead.
- When a connection is used, made dormant, and then restored to the current state in the same unit of work, that connection reflects its last use by the application process with regard to the status of locks, cursors, and prepared statements.

Examples:

Execute SQL statements at IBMSTHDB, execute SQL statements at IBMTOKDB, and then execute more SQL statements at IBMSTHDB.

```
EXEC SQL CONNECT TO IBMSTHDB;
/* Execute statements referencing objects at IBMSTHDB */
EXEC SQL CONNECT TO IBMTOKDB;
/* Execute statements referencing objects at IBMTOKDB */
EXEC SQL SET CONNECTION IBMSTHDB;
/* Execute statements referencing objects at IBMSTHDB */
```

Note that the first CONNECT statement creates the IBMSTHDB connection, the second CONNECT statement places it in the dormant state, and the SET CONNECTION statement returns it to the current state.

Related concepts:

- "Distributed relational databases" in the *SQL Reference, Volume 1*

Related reference:

- "CONNECT (Type 1)" on page 134

Related samples:

- "dbmcon.sqc -- How to use multiple databases (C)"
- "dbmcon.sqC -- How to use multiple databases (C++)"

SET CURRENT DEFAULT TRANSFORM GROUP

The SET CURRENT DEFAULT TRANSFORM GROUP statement changes the value of the CURRENT DEFAULT TRANSFORM GROUP special register. This statement is not under transaction control.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

No authorization is required to execute this statement.

Syntax:

Description:

group-name
Specifies a one-part name that identifies a transform group defined for all structured types. This name can be referenced in subsequent statements (or until the special register value is changed again using another SET CURRENT DEFAULT TRANSFORM GROUP statement).

The name must be an SQL identifier, up to 18 characters in length (SQLSTATE 42815). No validation that the *group-name* is defined for any structured type is made when the special register is set. Only when a structured type is specifically referenced is the definition of the named transform group checked for validity.

Rules:

- If the value specified does not conform to the rules for a *group-name*, an error is raised (SQLSTATE 42815)
- The TO SQL and FROM SQL functions defined in the *group-name* transform group are used for exchanging user-defined structured type data with a host program.

Notes:

- The initial value of the CURRENT DEFAULT TRANSFORM GROUP special register is the empty string.

Examples:

Example 1: Set the default transform group to MYSTRUCT1. The TO SQL and FROM SQL functions defined in the MYSTRUCT1 transform group will be used for exchanging user-defined structured type variables with the current host program.

```
SET CURRENT DEFAULT TRANSFORM GROUP = MYSTRUCT1
```

Related reference:

- "CURRENT DEFAULT TRANSFORM GROUP special register" in the *SQL Reference, Volume 1*

SET CURRENT DEGREE

The SET CURRENT DEGREE statement assigns a value to the CURRENT DEGREE special register. This statement is not under transaction control.

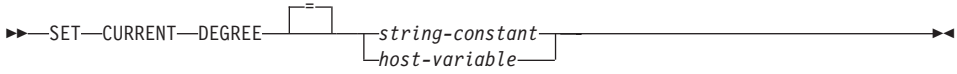
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. 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 value of the string constant or host variable. The value must be a character string that is not longer than 5 bytes. The value must be the character string representation of an integer between 1 and 32 767 inclusive or 'ANY'.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intra-partition parallelism.

If the value of CURRENT DEGREE is a number when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism with the specified degree.

If the value of CURRENT DEGREE is 'ANY' when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism using a degree determined by the database manager.

host-variable
The *host-variable* must be of data type CHAR or VARCHAR and the length must not exceed 5. If a longer field is provided, an error will be returned (SQLSTATE 42815). If the actual value provided is larger than the replacement value specified, the input must be padded on the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive. If a *host-variable* has an

associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

string-constant
The *string-constant* length must not exceed 5.

Notes:

The degree of intra-partition parallelism for static SQL statements can be controlled using the DEGREE option of the PREP or BIND command.

The actual runtime degree of intra-partition parallelism will be the lower of:
- Maximum query degree (max_querydegree) configuration parameter
- Application runtime degree
- SQL statement compilation degree

The intra_parallel database manager configuration must be on to use intra-partition parallelism. If it is set to off, the value of this register will be ignored and the statement will not use intra-partition parallelism for the purpose of optimization (SQLSTATE 01623).

Some SQL statements cannot use intra-partition parallelism.

Example:

Example 1: The following statement sets the CURRENT DEGREE to inhibit intra-partition parallelism.

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

Example 2: The following statement sets the CURRENT DEGREE to allow intra-partition parallelism.

```
SET CURRENT DEGREE = 'ANY'
```

SET CURRENT EXPLAIN MODE

The SET CURRENT EXPLAIN MODE statement changes the value of the CURRENT EXPLAIN MODE special register. It is not under transaction control.

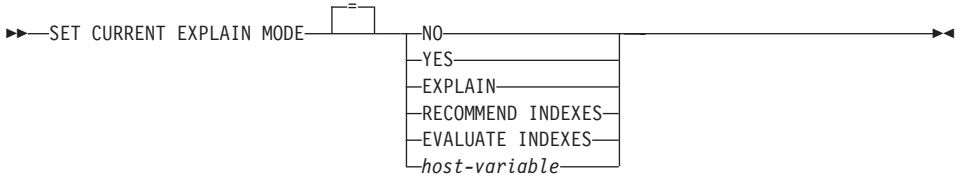
Invocation:

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization:

No special authorization is required to execute this statement.

Syntax:



Description:

NO
Disables the Explain facility. No Explain information is captured. NO is the initial value of the special register.

YES
Enables the Explain facility and causes Explain information to be inserted into the Explain tables for eligible dynamic SQL statements. All dynamic SQL statements are compiled and executed normally.

EXPLAIN
Enables the Explain facility and causes Explain information to be captured for any eligible dynamic SQL statement that is prepared. However, dynamic statements are not executed.

RECOMMEND INDEXES
Enables the SQL compiler to recommend indexes. All queries that are executed in this explain mode will populate the ADVISE_INDEX table with recommended indexes. In addition, Explain information will be captured in the Explain tables to reveal how the recommended indexes are used, but the statements are neither compiled nor executed.

EVALUATE INDEXES
Enables the SQL compiler to evaluate indexes. The indexes to be

evaluated are read from the ADVISE_INDEX table, and must be marked with EVALUATE = Y. The optimizer generates virtual indexes based on the values from the catalogs. All queries that are executed in this explain mode will be compiled and optimized using estimated statistics based on the virtual indexes. The statements are not executed.

host-variable
The *host-variable* must be of data type CHAR or VARCHAR and the length must not exceed 254. If a longer field is provided, an error will be returned (SQLSTATE 42815). The value specified must be NO, YES, EXPLAIN, RECOMMEND INDEXES, or EVALUATE INDEXES. If the actual value provided is larger than the replacement value specified, the input must be padded on the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive. If a *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Notes:

- Explain information for static SQL statements can be captured by using the EXPLAIN option of the PREP or BIND command. If the ALL value of the EXPLAIN option is specified, and the CURRENT EXPLAIN MODE register value is NO, explain information will be captured for dynamic SQL statements at run time. If the value of the CURRENT EXPLAIN MODE register is not NO, the value of the EXPLAIN bind option is ignored.
- RECOMMEND INDEXES and EVALUATE INDEXES are special modes which can only be set with the SET CURRENT EXPLAIN MODE statement. These modes cannot be set using PREP or BIND options, and they do not work with the SET CURRENT EXPLAIN SNAPSHOT statement.
- If the Explain facility is activated, the current authorization ID must have INSERT privilege for the Explain tables, or an error (SQLSTATE 42501) is raised.
- When SQL statements are explained from a routine, the routine must be defined with an SQL data access indicator of MODIFIES SQL DATA (SQLSTATE 42985).

Example:

The following statement sets the CURRENT EXPLAIN MODE special register, so that Explain information will be captured for any subsequent eligible dynamic SQL statements and the statement will not be executed.

```
SET CURRENT EXPLAIN MODE = EXPLAIN
```

Related reference:

- "Explain register values" in the *SQL Reference, Volume 1*

SET CURRENT EXPLAIN SNAPSHOT

The SET CURRENT EXPLAIN SNAPSHOT statement changes the value of the CURRENT EXPLAIN SNAPSHOT special register. It is not under transaction control.

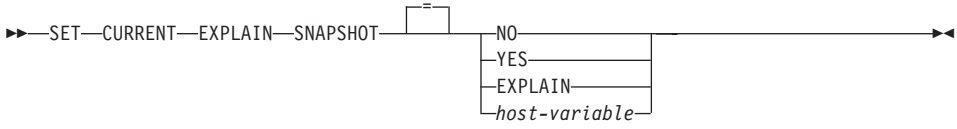
Invocation:

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:

NO
Disables the Explain snapshot facility. No snapshot is taken. NO is the initial value of the special register.

YES
Enables the Explain snapshot facility, creating a snapshot of the internal representation for each eligible dynamic SQL statement. This information is inserted in the SNAPSHOT column of the EXPLAIN_STATEMENT table.

The EXPLAIN SNAPSHOT facility is intended for use with Visual Explain.

EXPLAIN
Enables the Explain snapshot facility, creating a snapshot of the internal representation for each eligible dynamic SQL statement that is prepared. However, dynamic statements are not executed.

host-variable
The *host-variable* must be of data type CHAR or VARCHAR and the length of its contents must not exceed 8. If a longer field is provided, an error will be returned (SQLSTATE 42815). The value contained in this register must be either NO, YES, or EXPLAIN. If the actual value provided is larger than the replacement value specified, the input must be padded on

the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Notes:

- Explain snapshots for static SQL statements can be captured by using the EXPLSNAP option of the PREP or BIND command. If the ALL value of the EXPLSNAP option is specified, and the CURRENT EXPLAIN SNAPSHOT register value is NO, Explain snapshots will be captured for dynamic SQL statements at run time. If the value of the CURRENT EXPLAIN SNAPSHOT register is not NO, the EXPLSNAP option is ignored.
- If the Explain snapshot facility is activated, the current authorization ID must have INSERT privilege for the Explain tables or an error (SQLSTATE 42501) is raised.
- When SQL statements are explained from a routine, the routine must be defined with an SQL data access indicator of MODIFIES SQL DATA (SQLSTATE 42985).

Examples:

Example 1: The following statement sets the CURRENT EXPLAIN SNAPSHOT special register, so that an Explain snapshot will be taken for any subsequent eligible dynamic SQL statements and the statement will be executed.

```
SET CURRENT EXPLAIN SNAPSHOT = YES
```

Example 2: The following example retrieves the current value of the CURRENT EXPLAIN SNAPSHOT special register into the host variable called SNAP.

```
EXEC SQL VALUES (CURRENT EXPLAIN SNAPSHOT) INTO :SNAP;
```

Related reference:

- "Explain register values" in the *SQL Reference, Volume 1*
- "EXPLAIN_STATEMENT table" in the *SQL Reference, Volume 1*

SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

The SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement changes the value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register. It is not under transaction control.

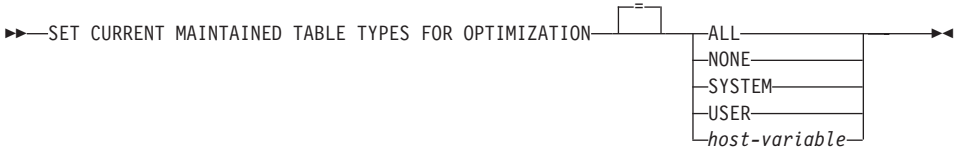
Invocation:

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:

ALL
Specifies that all possible types of maintained tables controlled by this special register, now and in the future, are to be considered when optimizing the processing of dynamic SQL queries.

NONE
Specifies that none of the object types that are controlled by this special register are to be considered when optimizing the processing of dynamic SQL queries.

SYSTEM
Specifies that system-maintained refresh-deferred materialized query tables can be considered to optimize the processing of dynamic SQL queries. (Immediate materialized query tables are always available.)

USER
Specifies that user-maintained refresh-deferred materialized query tables can be considered to optimize the processing of dynamic SQL queries.

host-variable
A variable of type CHAR or VARCHAR. The length of its content must not exceed 254 bytes (SQLSTATE 42815). The specified value must be ALL, NONE, SYSTEM, or USER. If the host variable has an associated

indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815). The characters of the host variable must be left justified. All input values are treated as being case-insensitive. The value must be padded on the right with blanks if its length is less than that of the host variable.

Notes:

- The initial value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register is SYSTEM.
- The CURRENT REFRESH AGE special register must be set to a value other than zero for the specified table types to be considered when optimizing the processing of dynamic SQL queries.

Examples:

Example 1: Set the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register.

```
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION SYSTEM = USER
```

Example 2: Retrieve the current value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register into a host variable called CURMAINTYPES.

```
EXEC SQL VALUES (CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION)
  INTO :CURMAINTYPES
```

Example 3: Set the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register to have no value.

```
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION = NONE
```

SET CURRENT PACKAGESET

The SET CURRENT PACKAGESET statement sets the schema name (collection identifier) that will be used to select the package to use for subsequent SQL statements. This statement is not under transaction control.

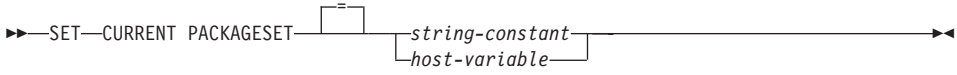
Invocation:

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared. This statement is not supported in REXX.

Authorization:

None required.

Syntax:



Description:

string-constant
A character string constant. If the value exceeds 30 bytes, only the first 30 bytes are used.

host-variable
A variable of type CHAR or VARCHAR. It cannot be set to null. If the value exceeds 30 bytes, only the first 30 bytes are used.

Notes:

- This statement allows an application to specify the schema name used when selecting a package for an executable SQL statement. The statement is processed at the client and does not flow to the application server.
- The COLLECTION bind option can be used to create a package with a specified schema name.
- Unlike DB2 UDB for OS/390 and z/OS, the SET CURRENT PACKAGESET statement is implemented without support for a special register called CURRENT PACKAGESET.

Example:

Assume an application called TRYIT is precompiled by user ID PRODUSA, making 'PRODUSA' the default schema name in the bind file. The application is then bound twice with different bind options. The following command line processor commands were used:

```
DB2 CONNECT TO SAMPLE USER PRODUSA
DB2 BIND TRYIT.BND DATETIME USA
DB2 CONNECT TO SAMPLE USER PRODEUR
DB2 BIND TRYIT.BND DATETIME EUR COLLECTION 'PRODEUR'
```

This creates two packages called TRYIT. The first bind command created the package in the schema named 'PRODUSA'. The second bind command created the package in the schema named 'PRODEUR' based on the COLLECTION option.

Assume the application TRYIT contains the following statements:

```
EXEC SQL CONNECT TO SAMPLE;
.
.
EXEC SQL SELECT HIREDATE INTO :HD FROM EMPLOYEE WHERE EMPNO='000010'; 1
.
.
EXEC SQL SET CURRENT PACKAGESET 'PRODEUR';                            2
.
.
EXEC SQL SELECT HIREDATE INTO :HD FROM EMPLOYEE WHERE EMPNO='000010'; 3
```

1 This statement will run using the PRODUSA.TRYIT package because it is the default package for the application. The date is therefore returned in USA format.

2 This statement sets the schema name to 'PRODEUR' for package selection.

3 This statement will run using the PRODEUR.TRYIT package as a result of the SET CURRENT PACKAGESET statement. The date is therefore returned in EUR format.

SET CURRENT QUERY OPTIMIZATION

The SET CURRENT QUERY OPTIMIZATION statement assigns a value to the CURRENT QUERY OPTIMIZATION special register. The value specifies the current class of optimization techniques enabled when preparing dynamic SQL statements. It is not under transaction control.

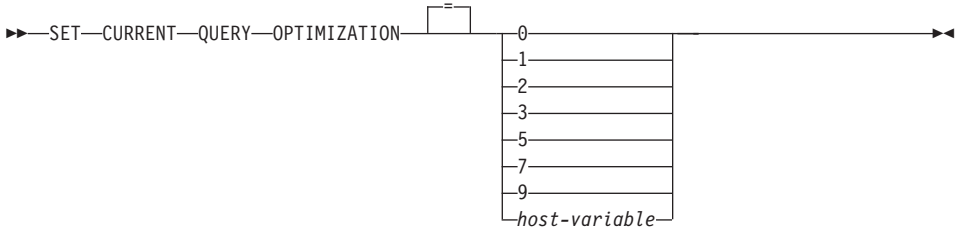
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization:

No authorization is required to execute this statement.

Syntax:



Description:

optimization-class

optimization-class can be specified either as an integer constant or as the name of a host variable that will contain the appropriate value at run time. An overview of the classes follows.

0 Specifies that a minimal amount of optimization is performed to generate an access plan. This class is most suitable for simple dynamic SQL access to well-indexed tables.

1 Specifies that optimization roughly comparable to DB2 Version 1 is performed to generate an access plan.

2 Specifies a level of optimization higher than that of DB2 Version 1, but at significantly less optimization cost than levels 3 and above, especially for very complex queries.

3 Specifies that a moderate amount of optimization is performed to generate an access plan.

5 Specifies a significant amount of optimization is performed to generate an access plan. For complex dynamic SQL queries, heuristic rules are used to limit the amount of time spent selecting an access plan. Where possible, queries will use materialized query tables instead of the underlying base tables.

7 Specifies a significant amount of optimization is performed to generate an access plan. Similar to 5 but without the heuristic rules.

9 Specifies a maximal amount of optimization is performed to generate an access plan. This can greatly expand the number of possible access plans that are evaluated. This class should be used to determine if a better access plan can be generated for very complex and very long-running queries using large tables. Explain and performance measurements can be used to verify that a better plan has been generated.

host-variable The data type is INTEGER. The value must be in the range 0 to 9 (SQLSTATE 42815) but should be 0, 1, 2, 3, 5, 7, or 9 (SQLSTATE 01608). If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Notes:

- When the CURRENT QUERY OPTIMIZATION register is set to a particular value, a set of query rewrite rules are enabled, and certain optimization variables take on particular values. This class of optimization techniques is then used during preparation of dynamic SQL statements.
- In general, changing the optimization class impacts the execution time of the application, the compilation time, and resources required. Most statements will be adequately optimized using the default query optimization class. Lower query optimization classes, especially classes 1 and 2, may be appropriate for dynamic SQL statements for which the resources consumed by the dynamic *PREPARE* are a significant portion of those required to execute the query. Higher optimization classes should be chosen only after considering the additional resources that may be consumed and verifying that a better access plan has been generated.
- Query optimization classes must be in the range 0 to 9. Classes outside this range will return an error (SQLSTATE 42815). Unsupported classes within this range will return a warning (SQLSTATE 01608) and will be replaced with the next lowest query optimization class. For example, a query optimization class of 6 will be replaced by 5.

- Dynamically prepared statements use the class of optimization that was set by the most recently executed SET CURRENT QUERY OPTIMIZATION statement. In cases where a SET CURRENT QUERY OPTIMIZATION statement has not yet been executed, the query optimization class is determined by the value of the database configuration parameter, dft_queryopt.
- Statically bound statements do not use the CURRENT QUERY OPTIMIZATION special register; therefore this statement has no effect on them. The QUERYOPT option is used during preprocessing or binding to specify the desired class of optimization for statically bound statements. If QUERYOPT is not specified then, the default value specified by the database configuration parameter, dft_queryopt, is used.
- The results of executing the SET CURRENT QUERY OPTIMIZATION statement are not rolled back if the unit of work in which it is executed is rolled back.

Examples:

Example 1: This example shows how the highest degree of optimization can be selected.

```
SET CURRENT QUERY OPTIMIZATION 9
```

Example 2: The following example shows how the CURRENT QUERY OPTIMIZATION special register can be used within a query.

Using the SYSCAT.PACKAGES catalog view, find all plans that were bound with the same setting as the current value of the CURRENT QUERY OPTIMIZATION special register.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT PKGNAME, PKGSCHEMA FROM SYSCAT.PACKAGES
  WHERE QUERYOPT = CURRENT QUERY OPTIMIZATION
```

SET CURRENT REFRESH AGE

The SET CURRENT REFRESH AGE statement changes the value of the CURRENT REFRESH AGE special register. It is not under transaction control.

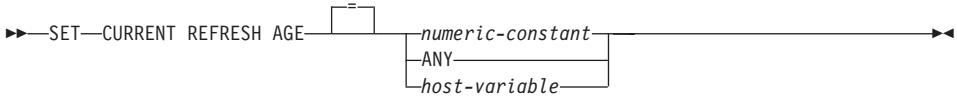
Invocation:

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:

numeric-constant
A DECIMAL(20,6) value representing a timestamp duration. The value must be 0 or 99 999 999 999 999 (the microseconds portion of the value is ignored and can therefore be any value).

0	Indicates that only materialized query tables defined with REFRESH IMMEDIATE may be used to optimize the processing of a query.
99999999999999	Indicates that any table types specified by CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION, and materialized query tables defined with REFRESH IMMEDIATE, can be used to optimize the processing of a query. This value represents 9 999 years, 99 months, 99 days, 99 hours, 99 minutes, and 99 seconds.

ANY
This is a shorthand for 99999999999999.

host-variable
A variable of type DECIMAL(20,6) or other type that is assignable to DECIMAL(20,6). It cannot be set to null. If *host-variable* has an associated

indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815). The value of the host-variable must be 0 or 99 999 999 999 999.000000.

Notes:

- The initial value of the CURRENT REFRESH AGE special register is zero.
- Setting the CURRENT REFRESH AGE special register to a value other than zero should be done with caution. By allowing a table (of a type specified by the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register) that may not represent the values of the underlying base table to be used to optimize the processing of a query, the result of the query may *not* accurately represent the data in the underlying table. This may be reasonable if you know that the underlying data has not changed, or you are willing to accept a degree of error in the results based on your knowledge of the cached data.
- The CURRENT REFRESH AGE value of 99 999 999 999 999 cannot be used in timestamp arithmetic operations since the result would be outside the valid range of dates (SQLSTATE 22008).

Examples:

Example 1: The following statement sets the CURRENT REFRESH AGE special register.

```
SET CURRENT REFRESH AGE ANY
```

Example 2:

The following example retrieves the current value of the CURRENT REFRESH AGE special register into the host variable called CURMAXAGE.

```
EXEC SQL VALUES (CURRENT REFRESH AGE) INTO  :CURMAXAGE;
```

The value would be 99999999999999.000000, set by the previous example.

SET ENCRYPTION PASSWORD

The SET ENCRYPTION PASSWORD statement sets the password that will be used by the ENCRYPT, DECRYPT_BIN and DECRYPT_CHAR functions. The password is not tied to DB2 authentication, and is used for data encryption and decryption only.

This statement is not under transaction control.

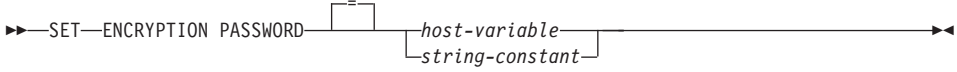
Invocation:

The 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 ENCRYPTION PASSWORD can be used by the ENCRYPT, DECRYPT_BIN, and DECRYPT_CHAR built-in functions for password based encryption. The length must be between 6 and 127 bytes. All characters must be specified in the exact case intended as there is no automatic conversion to uppercase characters.

host-variable
A variable of type CHAR or VARCHAR. The length of the *host-variable* must be between 6 and 127 bytes (SQLSTATE 428FC). It cannot be set to null. All characters are specified in the exact case intended, as there is no conversion to uppercase characters.

string-constant
A character string constant. The length must be between 6 and 127 bytes (SQLSTATE 428FC).

Notes:

- The initial value of the ENCRYPTION PASSWORD is the empty string (''').
- The *host-variable* or *string-constant* is transmitted to the database server using normal DB2 mechanisms.

Examples:

Example 1: The following statement sets the ENCRYPTION PASSWORD.

```
SET ENCRYPTION PASSWORD = 'Gre89Ea'
```

Related reference:

- "DECRYPT_BIN and DECRYPT_CHAR scalar functions" in the *SQL Reference, Volume 1*
- "ENCRYPT scalar function" in the *SQL Reference, Volume 1*

SET EVENT MONITOR STATE

The SET EVENT MONITOR STATE statement activates or deactivates an event monitor. The current state of an event monitor (active or inactive) is determined by using the EVENT_MON_STATE built-in function. The SET EVENT MONITOR STATE statement is not under transaction control.

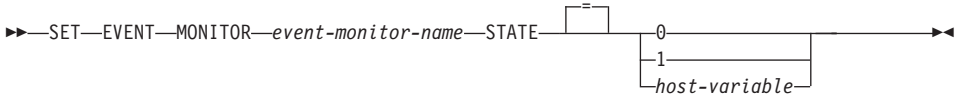
Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The authorization ID of the statement most hold either SYSADM or DBADM authority (SQLSTATE 42815).

Syntax:



Description:

event-monitor-name
Identifies the event monitor to activate or deactivate. The name must identify an event monitor that exists in the catalog (SQLSTATE 42704).

new-state
new-state can be specified either as an integer constant or as the name of a host variable that will contain the appropriate value at run time. The following may be specified:

0 Indicates that the specified event monitor should be deactivated.

1 Indicates that the specified event monitor should be activated. The event monitor should not already be active; otherwise a warning (SQLSTATE 01598) is issued.

host-variable The data type is INTEGER. The value specified must be 0 or 1 (SQLSTATE 42815). If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Rules:

- Although an unlimited number of event monitors may be defined, there is a limit of 32 event monitors that can be simultaneously active (SQLSTATE 54030).
- In order to activate an event monitor, the transaction in which the event monitor was created must have been committed (SQLSTATE 55033). This rule prevents (in one unit of work) creating an event monitor, activating the monitor, then rolling back the transaction.
- If the number or size of the event monitor files exceeds the values specified for MAXFILES or MAXFILESIZE on the CREATE EVENT MONITOR statement, an error (SQLSTATE 54031) is raised.
- If the target path of the event monitor (that was specified on the CREATE EVENT MONITOR statement) is already in use by another event monitor, an error (SQLSTATE 51026) is raised.

Notes:

- Activating an event monitor performs a reset of any counters associated with it.

Example:

The following example activates an event monitor called SMITHPAY.

```
SET EVENT MONITOR SMITHPAY STATE = 1
```

SET INTEGRITY

The SET INTEGRITY statement is used to:

- Turn off integrity checking for one or more tables. This includes check constraint and referential constraint checking, DATALINK integrity checking, and generation of values for generated columns. If the table is a materialized query table defined with REFRESH IMMEDIATE, or a staging table with the PROPAGATE IMMEDIATE attribute, immediate refreshing of the data is turned off. This puts the table in *check pending state*, in which only limited access by a restricted set of statements and commands is allowed. Primary key and unique constraints continue to be checked.
- Turn integrity checking back on and carry out all deferred checking for one or more tables. If the table is a system-maintained materialized query table or a staging table, the data is refreshed as necessary. If the materialized query table is defined with the REFRESH IMMEDIATE attribute, or the staging table is defined with the PROPAGATE IMMEDIATE attribute, immediate refreshing of the data is turned on. The descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables for tables that were put into check pending state using the CASCADE DEFERRED option are put (if necessary) into check pending no access state.
- Turn integrity checking on for one or more tables without first carrying out any deferred integrity checking. If the table is a materialized query table defined with the REFRESH IMMEDIATE attribute, or a staging table defined with the PROPAGATE IMMEDIATE attribute, immediate refreshing of the data is turned on. The descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables for tables that were put into check pending state using the CASCADE DEFERRED option are put (if necessary) into check pending no access state.
- Place the table in check pending state if the table is already in DataLink Reconcile Pending (DRP) or DataLink Reconcile Not Possible (DRNP) state. If a table is not in either of those states, then unconditionally set the table to DRP state and check pending state.
- Put one or more tables from the no data movement mode back to the full access mode.
- Prune the contents of one or more staging tables.

When the statement is used to check integrity for a table after it has been loaded, the system can incrementally process the table by checking only the appended portion for constraint violations. If the subject table is a materialized query table or a staging table, and the load operations are performed on its underlying tables, the system can incrementally refresh the materialized query table or incrementally propagate the staging table with

only the appended portions of its underlying tables. However, there are some situations in which the system will not be able to perform such optimizations and will instead fully process to ensure data integrity. Full processing is done by checking the entire table for constraints violations, recomputing a materialized query table's definition, or marking a staging table as inconsistent. The latter implies that a full refresh of its associated materialized query table is required. There is also a situation in which the user may want to explicitly request incremental processing by specifying the INCREMENTAL option.

The SET INTEGRITY statement is under transaction control.

Invocation:

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization:

The privileges required to execute SET INTEGRITY depend on the use of the statement, as outlined below:

- Turn off integrity checking.

  The privileges of the authorization ID of the statement must include at least one of the following:

  – CONTROL privilege on:
    - The specified tables, and
    - The descendent foreign key tables that will have integrity checking turned off by the statement, and
    - The descendent immediate materialized query tables that will have integrity checking turned off by the statement, and
    - The descendent immediate staging tables that will have integrity checking turned off by the statement.
  – SYSADM or DBADM authority
  – LOAD authority
- Both turn on integrity checking and carry out checking.

  The privileges of the authorization ID of the statement must include at least one of the following:

  – SYSADM or DBADM authority
  – CONTROL privilege on the tables that are being checked *and*, if exceptions are being posted to one or more tables, INSERT privilege on the exception tables. CONTROL privilege on all descendent foreign key

tables, descendent immediate materialized query tables, and descendent immediate staging tables that will implicitly be placed in check pending state by the statement.

– LOAD authority and, if exceptions are being posted to one or more tables:
  - SELECT and DELETE privilege on each table being checked; and
  - INSERT privilege on the exception tables.

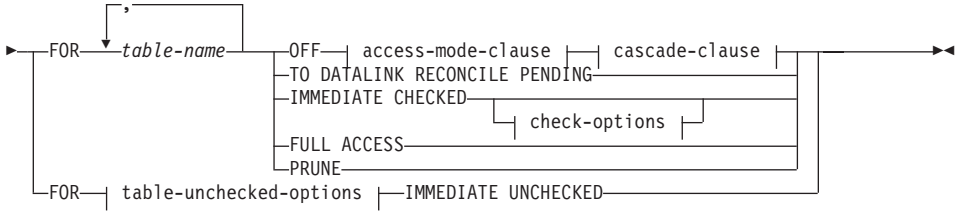
- Turn on integrity checking without first carrying out checking.

  The authorization ID of the statement must have at least one of the following:
  – SYSADM or DBADM authority
  – CONTROL privilege on the tables that are being checked. CONTROL privilege on each descendent foreign key table, descendent immediate materialized query table, and descendent immediate staging table that will implicitly be placed in check pending state by the statement
  – LOAD authority
- Bring the table from no data movement mode to full access mode.

  The authorization ID of the statement must have at least one of the following:
  – SYSADM or DBADM authority
  – CONTROL privilege on the tables that are moving from no data movement mode to full access mode.
  – LOAD authority
- Prune a staging table.

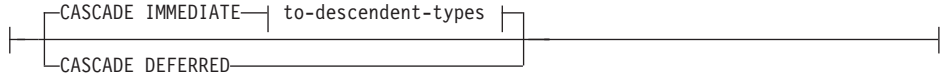
  The authorization ID of the statement must have at least one of the following:
  – SYSADM or DBADM authority
  – CONTROL privilege on the table being pruned.

Syntax:

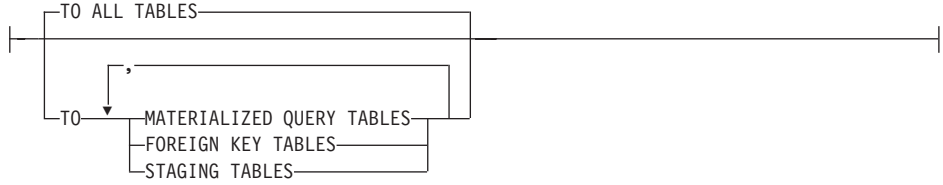


access-mode-clause:

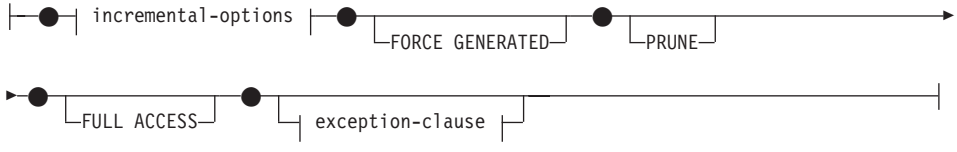
cascade-clause:



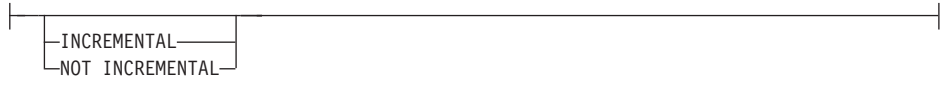
to-descendent-types:



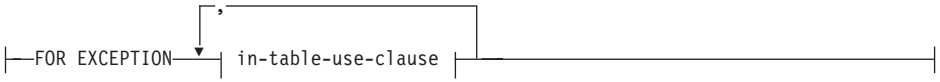
check-options:



incremental-options:

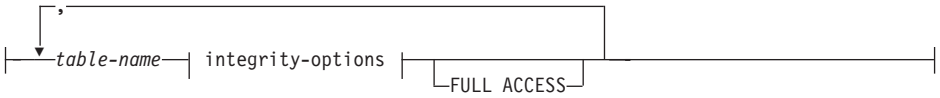


exception-clause:

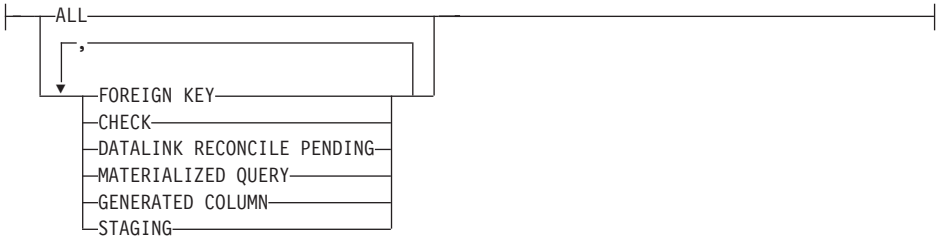


in-table-use-clause:

table-unchecked-options:



integrity-options:



Description:

FOR *table-name*

Identifies one or more tables for integrity processing. It must be a table described in the catalog and must not be a view, catalog table, or typed table.

OFF

Specifies that the tables are to have their foreign key constraints, check constraints, and column generation turned off and are, therefore to be placed in check pending state. If it is a materialized query table or a staging table, immediate refreshing is turned off (if applicable), and the materialized query table or the staging table is placed in check pending state.

Note that it is possible that a table may already be in check pending state with only one type of integrity checking turned off; in such a situation, the other type of integrity checking will also be turned off.

Only very limited activity is allowed on a table that is in check pending state.

access-mode-clause
Specifies the readability of the table while it is in check pending state.

NO ACCESS
Specifies that the table is to be put in check pending no access state, which does not allow read or write access to the table.

READ ACCESS
Specifies that the table is to be put in check pending read state, which allows read access to the non-appended portion of the table. This option is not allowed on a table that is in check pending no access state (SQLSTATE 428FH).

If the *access-mode-clause* is not specified, the table is put in check pending no access state.

cascade-clause
Specifies whether the check pending state of the table referenced in the SET INTEGRITY statement is to be immediately cascaded to all descendent foreign key tables, all descendent immediate materialized query tables, and all descendent immediate staging tables.

CASCADE IMMEDIATE
Specifies that the check pending state for foreign key constraints is to be immediately extended to all descendent foreign key tables. If the table has descendent immediate materialized query tables or descendent immediate staging tables, the check pending state is extended immediately to the materialized query tables and the staging tables.

When the table is later checked for integrity violations, and is taken out of check pending state, all descendent tables that are in check pending read state will be put in check pending no access state.

to-descendent-types

TO ALL TABLES
Specifies that the check pending state is to be immediately cascaded to all descendent tables of the tables in the invocation list. Descendent tables include all descendent foreign key tables, immediate staging tables, and immediate materialized query tables that are descendants of the tables in the invocation list, or descendants of descendent foreign key tables.

Specifying TO ALL TABLES is equivalent to specifying TO FOREIGN KEY TABLES, TO MATERIALIZED QUERY TABLES, *and* TO STAGING TABLES, all in the same statement.

TO MATERIALIZED QUERY TABLES
If only TO MATERIALIZED QUERY TABLES is specified, the

check pending state is to be immediately cascaded only to descendent immediate materialized query tables. Other descendent tables may later be put in check pending state, if necessary, when the table is brought out of check pending state. If both TO FOREIGN KEY TABLES and TO MATERIALIZED QUERY TABLES are specified, the check pending state will be immediately cascaded to all descendent foreign key tables, all descendent immediate materialized query tables of the tables in the invocation list, and all immediate materialized query tables that are descendants of the descendent foreign key tables.

TO FOREIGN KEY TABLES
Specifies that the check pending state is to be immediately cascaded to descendent foreign key tables. Other descendent tables may later be put in check pending state, if necessary, when the table is brought out of check pending state.

TO STAGING TABLES
Specifies that the check pending state is to be immediately cascaded to descendent staging tables. Other descendent tables may later be put in check pending state, if necessary, when the table is brought out of check pending state. If both TO FOREIGN KEY TABLES and TO STAGING TABLES are specified, the check pending state will be immediately cascaded to all descendent foreign key tables, all descendent immediate staging tables of the tables in the invocation list, and all immediate staging tables that are descendants of the descendent foreign key tables.

CASCADE DEFERRED
Specifies that only the tables in the invocation list are to be put in check pending state. The states of the descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables will remain unchanged. Descendent foreign key tables may later be implicitly put in check pending no access state when their parent tables are checked for constraints violations (using the IMMEDIATE CHECKED option of the SET INTEGRITY statement). Descendent immediate materialized query tables and descendent immediate staging tables may be implicitly put in check pending no access state when one of their underlying tables is checked for integrity violations. A warning (SQLSTATE 01586) will be issued to indicate that descendent tables have been put in check pending state.

If the *cascade-clause* is not specified, the check pending state is immediately cascaded to dependent and descendent tables.

TO DATALINK RECONCILE PENDING
Specifies that the tables are to have DATALINK integrity checking turned

off, and that the tables are to be put in check pending no access state. If the table is already in DataLink Reconcile Not Possible (DRNP) state, it remains in this state with check pending. Otherwise, the table is set to DataLink Reconcile Pending (DRP) state.

Dependent and descendent table are not affected when this option is specified.

IMMEDIATE CHECKED
Specifies that the table is to have its integrity checking turned on and that the integrity checking that was deferred is to be carried out. This is done in accordance with the information set in the STATUS and CONST_CHECKED columns of the SYSCAT.TABLES catalog. That is:

- The value in STATUS must be C (the table is in check pending state), or an error (SQLSTATE 51027) is returned, unless the table is a descendent foreign key table, descendent materialized query table, or descendent staging table of a table that is specified in the list, is in check pending state, and whose intermediate ancestors are also in the list.
- If the table being checked is in check pending state, the value in CONST_CHECKED indicates which integrity options are to be checked.

If the table was put in check pending state using the CASCADE DEFERRED option, its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables are, if necessary, put in check pending no access state. A warning (SQLSTATE 01586) is issued to indicate that descendent tables have been put in check pending state.

If it is a system-maintained materialized query table, the data is checked against the query and refreshed as necessary. (This statement cannot be used for user-maintained materialized query tables.) If it is a staging table, the data is checked against its query definition and propagated as necessary.

When the integrity of a child table is checked:

- None of its parents can be in check pending state, or
- Each of its parents must be checked for constraints violations in the same SET INTEGRITY statement.

When an immediate materialized query table is refreshed, or deltas are propagated to a staging table:

- None of its underlying tables can be in check pending state, or
- Each of its underlying tables must be checked in the same SET INTEGRITY statement.

Otherwise, an error is returned (SQLSTATE 428A8).

DATALINK values are not checked, even if the table is in DRP or DRNP state. The RECONCILE command or API should be used to perform the reconciliation of DATALINK values. The table is taken out of check pending state, but continues to have the DRP or DRNP flag set. The table is usable, because the reconciliation of DATALINK values is deferred.

check-options

incremental-options

INCREMENTAL
Specifies the application of integrity checks on the appended portion (if any) of the table. If such a request cannot be satisfied (that is, the system detects that the whole table needs to be checked for data integrity), an error (SQLSTATE 55019) is returned.

NOT INCREMENTAL
Specifies the application of integrity checks on the whole table. If the table is a materialized query table, the materialized query table definition is recomputed. If the table has at least one constraint defined on it, this option forces full processing of descendent foreign key tables and descendent immediate materialized query tables. If the table is a staging table, it is set to inconsistent state.

If the *incremental-options* clause is not specified, the system will determine if incremental processing is possible; if not, the whole table will be checked.

FORCE GENERATED
If the table includes generated columns, the values are computed on the basis of the expression and stored in the column. If this clause is not specified, the current values are compared to the computed value of the expression, as if an equality check constraint existed. If the table is checked for integrity incrementally, generated columns will be computed only for the appended portion.

PRUNE
This option can be specified for staging tables only. Specifies that the content of the staging table is to be pruned, and that the staging table is to be set to an inconsistent state. If any table in the *table-name* list is not a staging table, an error is returned (SQLSTATE 428FH). If the INCREMENTAL check option is also specified, an error is returned (SQLSTATE 428FH).

FULL ACCESS
Specifies that the tables are to become fully accessible after the

SET INTEGRITY statement executes. This option can be specified in both the IMMEDIATE CHECKED and IMMEDIATE UNCHECKED clause.

When an underlying table in the invocation list is incrementally processed, and has dependent immediate materialized query tables or dependent immediate staging tables, the underlying table will be placed, as required, in the no data movement mode after the SET INTEGRITY statement. When all incrementally refreshable dependent immediate materialized query tables and staging tables are taken out of check pending state, the underlying table will automatically be brought out of the no data movement mode into the full access mode. If the FULL ACCESS option is specified in the IMMEDIATE CHECKED or IMMEDIATE UNCHECKED clause, the underlying table will bypass the no data movement mode and go directly into the full access mode. Dependent immediate materialized query tables that have not been refreshed may undergo a full recomputation in the subsequent REFRESH statement, and dependent immediate staging tables that have not had the appended portions of the table propagated to them may be flagged as inconsistent.

When an underlying table in the invocation list requires full processing, or does not have dependent immediate materialized query tables, or dependent immediate staging tables, the underlying table will go directly into the full access mode after the SET INTEGRITY statement, regardless of whether the FULL ACCESS option is specified.

Specifying the FULL ACCESS option in the IMMEDIATE UNCHECKED clause will result in an error (SQLSTATE 428FH) if the statement does not bring the table out of check pending state.

exception-clause

FOR EXCEPTION

Indicates that any row that is in violation of a foreign key constraint or a check constraint will be copied to an exception table and deleted from the original table. Even if errors are detected the constraints are turned back on again, and the table is taken out of check pending state. A warning (SQLSTATE 01603) is issued to indicate that one or more rows have been moved to the exception tables.

If the FOR EXCEPTION clause is not specified and any constraints are violated, only the first violation detected is returned to the user (SQLSTATE 23514). In the case of a

violation in any table, all of the tables are left in check pending state, as they were before the execution of the statement.

IN *table-name*
Specifies the table from which rows that violate constraints are to be copied. There must be one exception table specified for each table being checked. This clause cannot be specified if the table is a materialized query table or a staging table (SQLSTATE 428A7).

USE *table-name*
Specifies the exception table into which error rows are to be copied.

FULL ACCESS
If the FULL ACCESS option is specified as the only operation of the statement, the table will be brought out of the no data movement mode into the full access mode. The table is not rechecked for integrity violations. However, dependent immediate materialized query tables that have not been refreshed may require a full recomputation in subsequent REFRESH statements, and dependent immediate staging tables that have not had the appended portions of the table propagated to them may be changed to incomplete state. This option can only be specified for a table that is in the no data movement mode (SQLSTATE 428FH).

PRUNE
This option can be specified for staging tables only. Specifies that the content of the staging table is to be pruned, and that the staging table is to be set to an inconsistent state. If any table in the *table-name* list is not a staging table, an error is returned (SQLSTATE 428FH).

table-unchecked-options

table-name
Identifies one or more tables for integrity processing. It must be a table described in the catalog and must not be a view, catalog table, or typed table.

integrity-options
Used to define the integrity options that are set to IMMEDIATE UNCHECKED.

ALL
All integrity options will be turned on, and the table will be brought out of check pending state.

FOREIGN KEY
Foreign key constraints will be turned on when the table is brought out of check pending state.

CHECK
Check constraints will be turned on when the table is brought out of check pending state.

DATALINK RECONCILE PENDING
DATALINK integrity constraints will be turned on when the table is brought out of check pending state.

MATERIALIZED QUERY
Immediate refreshing will be turned on for a materialized query table with the REFRESH IMMEDIATE attribute.

GENERATED COLUMN
Generated columns will be turned on when the table is brought out of check pending state.

STAGING
Immediate propagation will be turned on for the staging table.

FULL ACCESS
Specifies that the tables are to become fully accessible after the SET INTEGRITY statement executes. This option can be specified in both the IMMEDIATE CHECKED and IMMEDIATE UNCHECKED clause.

When an underlying table in the invocation list is incrementally processed, and has dependent immediate materialized query tables or dependent immediate staging tables, the underlying table will be placed, as required, in the no data movement mode after the SET INTEGRITY statement. When all incrementally refreshable dependent immediate materialized query tables and staging tables are taken out of check pending state, the underlying table will automatically be brought out of the no data movement mode into the full access mode. If the FULL ACCESS option is specified in the IMMEDIATE CHECKED or IMMEDIATE UNCHECKED clause, the underlying table will bypass the no data movement mode and go directly into the full access mode. Dependent immediate materialized query tables that have not been refreshed may undergo a full recomputation in the subsequent REFRESH statement, and dependent immediate staging tables that have not had the appended portions of the table propagated to them may be flagged as inconsistent.

When an underlying table in the invocation list requires full processing, or does not have dependent immediate materialized query tables, or dependent immediate staging tables, the underlying table will go directly into the full access mode after the SET INTEGRITY statement, regardless of whether the FULL ACCESS option is specified.

Specifying the FULL ACCESS option in the IMMEDIATE UNCHECKED clause will result in an error (SQLSTATE 428FH) if the statement does not bring the table out of check pending state.

IMMEDIATE UNCHECKED

Specifies one of the following:

- The table is to have its integrity checking turned on (and, therefore, is to be taken out of check pending state) without being checked for integrity violations.

  This is indicated either by specifying ALL, or by specifying one or more of the other *integrity-options* for each integrity option that is off for that table.

- The table is to have one type of integrity checking turned on without being checked for this type of integrity violation, but is to be left in check pending state.

  This is indicated by specifying one or more of CHECK, FOREIGN KEY, MATERIALIZED QUERY, STAGING, GENERATED COLUMN, or DATALINK RECONCILE PENDING, such that at least one integrity option remains off for that table.

The implications with respect to data integrity should be considered before using this option. See the "Notes" section of this statement.

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - SET CONSTRAINTS can be specified in place of SET INTEGRITY
    - SUMMARY can be specified in place of MATERIALIZED QUERY
- Effects on tables in one of the check pending states:
  - Use of INSERT, UPDATE, or DELETE is disallowed on a table that is in check pending read or no access state. Furthermore, any statement that requires such a modification to a table that is in check pending state will be rejected.

    For example, deletion of a row in a parent table that cascades to a dependent table that is in check pending state is not allowed.
  - Use of SELECT is disallowed on a table that is in check pending no access state. Furthermore, any statement that requires read access to a table that is in check pending no access state will be rejected.
  - New constraints added to a table are normally enforced immediately. However, if the table is in check pending state, the checking of any new constraints is deferred until the table is taken out of check pending state. If the table is in check pending read state, addition of a new constraint forces the table into check pending no access state, because validity of data is at risk.

  - The CREATE INDEX statement cannot reference any table that is in check pending state. Similarly, an ALTER TABLE statement to add a primary key or a unique constraint cannot reference any table that is in check pending state.
  - The IMPORT utility is not allowed to operate on a table that is in check pending state. (The IMPORT utility differs from the LOAD utility in that it always checks constraints immediately.)
  - The EXPORT utility is not allowed to operate on a table that is in check pending no access state, but is allowed to operate on a table that is in check pending read state. If a table is in check pending read state, the EXPORT utility will only export the data that is in the non-appended portion.
  - Operations (like REORG, REDISTRIBUTE, update partitioning key, update clustering key, and so on) that may involve data movement within a table are not allowed to operate on a table that is in any of the check pending states, or in the no data movement mode.
  - The utilities LOAD, BACKUP, RESTORE, UPDATE STATISTICS, RUNSTATS, REORGCHK, LIST HISTORY, and ROLLFORWARD are allowed on a table that is in any of the check pending states.
  - The statements ALTER TABLE, COMMENT, DROP TABLE, CREATE ALIAS, CREATE TRIGGER, CREATE VIEW, GRANT, REVOKE, and SET INTEGRITY can reference a table that is in any of the check pending states. However, they may cause the table to be put into the no access mode.
  - Packages, views and any other objects that depend on a table that is in check pending no access state will return an error when the table is accessed at run time. Packages that depend on a table that is in check pending read state will return an error when an insert, update, or delete operation is attempted on the table at run time.

  The removal of violating rows by the SET INTEGRITY statement is not a delete event. Therefore, triggers are never activated by a SET INTEGRITY statement. Similarly, updating generated columns using the FORCE GENERATED option does not activate triggers.

- Incremental processing will be used whenever the situation allows it, because it is more efficient. The INCREMENTAL option is not needed in most cases. It is needed, however, to ensure that integrity checks are indeed processed incrementally. If the system detects that full processing is needed to ensure data integrity, an error (SQLSTATE 55019) is returned.
- Warning about the use of the IMMEDIATE UNCHECKED clause:
  - This clause is intended to be used by utility programs, and its use by application programs is not recommended. If there is data in the table

that does not meet the integrity specifications that were defined for the table, and the IMMEDIATE UNCHECKED clause is used, incorrect query results may be returned.

The fact that integrity checking was turned on without doing deferred checking will be recorded in the catalog (the value in the CONST_CHECKED column in the SYSCAT.TABLES view will be set to 'U'). This indicates that the user has assumed responsibility for data integrity with respect to the specific constraints. This value remains unchanged until either:

- The table is put back into check pending state (by referencing the table in a SET INTEGRITY statement with the OFF clause), at which time 'U' values in the CONST_CHECKED column are changed to 'W' values, indicating that the user had previously assumed responsibility for data integrity, and the system needs to verify the data.
- All unchecked constraints for the table are dropped.

The 'W' state differs from the 'N' state in that it records the fact that integrity was previously checked by the user, but not yet by the system. If the user issues the SET INTEGRITY ... IMMEDIATE CHECKED statement with the NOT INCREMENTAL option, the system rechecks the whole table for data integrity (or performs a full refresh on a materialized query table), and then changes the 'W' state to the 'Y' state. If IMMEDIATE UNCHECKED is specified, or if NOT INCREMENTAL is not specified, the 'W' state is changed back to the 'U' state to record the fact that some data has still not been verified by the system. In the latter case (when the NOT INCREMENTAL is not specified), a warning (SQLSTATE 01636) is returned.

If an underlying table's integrity has been checked using the IMMEDIATE UNCHECKED clause, the 'U' values in the CONST_CHECKED column of the underlying table will be propagated to the corresponding CONST_CHECKED column of:

- Dependent immediate materialized query tables
- Dependent deferred materialized query tables
- Dependent staging tables

For a dependent immediate materialized query table, this propagation is done whenever the underlying table is brought out of check pending state, and whenever the materialized query table is refreshed. For a dependent deferred materialized query table, this propagation is done whenever the materialized query table is refreshed. For dependent staging tables, this propagation is done whenever the underlying table is brought out of check pending state. These propagated 'U' values in the CONST_CHECKED columns of dependent materialized query tables and

staging tables record the fact that these materialized query tables and staging tables depend on some underlying table whose integrity has been checked using the IMMEDIATE UNCHECKED clause.

For a materialized query table, the 'U' value in the CONST_CHECKED column that was propagated by the underlying table will remain until the materialized query table is fully refreshed and none of its underlying tables have a 'U' value in their corresponding CONST_CHECKED column. After such a refresh, the 'U' value in the CONST_CHECKED column for the materialized query table will be changed to 'Y'.

For a staging table, the 'U' value in the CONST_CHECKED column that was propagated by the underlying table will remain until the corresponding deferred materialized query table of the staging table is refreshed. After such a refresh, the 'U' value in the CONST_CHECKED column for the staging table will be changed to 'Y'.

- If a child table and its parent table are checked in the same SET INTEGRITY ... IMMEDIATE CHECKED statement, and the parent table requires full checking of its constraints, the child table will have its foreign key constraints checked, independently of whether or not the child table has a 'U' value in the CONST_CHECKED column for foreign key constraints.

- After appending data using LOAD INSERT, the SET INTEGRITY ... IMMEDIATE CHECKED statement checks the table for constraints violations. The system determines whether incremental processing on the table is possible. If so, only the appended portion is checked for integrity violations. If not, the system checks the whole table for integrity violations.
- Consider the statement:

  ```
  SET INTEGRITY FOR T IMMEDIATE CHECKED
  ```

  Situations in which the system will require a full refresh, or will check the whole table for integrity (the INCREMENTAL option cannot be specified) are:
  - When new constraints have been added to T itself
  - When a LOAD REPLACE operation against T, it parents, or its underlying tables has taken place
  - When the NOT LOGGED INITIALLY WITH EMPTY TABLE option has been activated after the last integrity check on T, its parents, or its underlying tables
  - The cascading effect of full processing, when any parent of T (or underlying table, if T is a materialized query table or a staging table) has been checked for integrity non-incrementally
  - If the table was in check pending state before migration, full processing is required the first time the table is checked for integrity after migration

  - If the table space containing the table or its parent (or underlying table of a materialized query table or a staging table) has been rolled forward to a point in time, and the table and its parent (or underlying table if the table is a materialized query table or a staging table) reside in different table spaces
  - When T is a materialized query table, and a LOAD REPLACE or LOAD INSERT operation directly into T has taken place after the last refresh
- If the conditions for full processing described in the previous bullet are not satisfied, the system will attempt to check only the appended portion for integrity, or perform an incremental refresh (if it is a materialized query table) when the user does not specify the NOT INCREMENTAL option for the statement `SET INTEGRITY FOR T IMMEDIATE CHECKED`.
- A table that is in DataLink Reconcile Not Possible (DRNP) state requires corrective action to be taken (possibly outside of the database). Once corrective action is completed, the table is taken out of DRNP state using the IMMEDIATE UNCHECKED option. The RECONCILE command or API should then be used to check the DATALINK integrity constraints.
- While integrity is being checked, an exclusive lock is held on each table specified during SET INTEGRITY invocation. A shared lock is acquired on each table that is not specified during SET INTEGRITY invocation, but that is a parent or underlying table of one of the dependent tables being checked.
- If an error occurs during integrity checking, all the effects of the checking (including deleting from the original and inserting into the exception tables) will be rolled back.
- If a SET INTEGRITY statement issued with the FORCE GENERATED option fails because of a lack of log space, increase available active log space and reissue the SET INTEGRITY statement. Alternatively, use the SET INTEGRITY GENERATED COLUMN IMMEDIATE UNCHECKED statement to bypass generated column checking for the table. Then, issue a SET INTEGRITY IMMEDIATE CHECKED statement without the FORCE GENERATED option to check the table for other integrity violations (if applicable) and to bring it out of check pending state. Once the table is out of check pending state, the generated columns can be updated to their default (generated) values by assigning them to the keyword DEFAULT in an UPDATE statement. This is accomplished by using either multiple searched update statements based on ranges (each followed by a commit), or a cursor-based approach using intermittent commits. A “with hold” cursor should be used if locks are to be retained after intermittent commits using the cursor-based approach.
- A table that was put into check pending state using the CASCADE DEFERRED option (of the SET INTEGRITY statement or the LOAD command), and that is checked for integrity violations using the IMMEDIATE CHECKED option of the SET INTEGRITY statement, will

have its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables put in check pending no access state, as required:

- If the entire table is checked for integrity violations, its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables will be put in check pending no access state.
- If the table is checked for integrity violations incrementally, its descendent immediate materialized query tables and staging tables will be put in check pending no access state, and its descendent foreign key tables will remain in their original states.
- If the table requires no checking at all, its descendent immediate materialized query tables, descendent staging tables, and descendent foreign key tables will remain in their original states.

- A table that was put in check pending state using the CASCADE DEFERRED option (of the SET INTEGRITY statement or the LOAD command), and that is brought out of check pending state using the IMMEDIATE UNCHECKED option of the SET INTEGRITY statement, will have its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables put in check pending no access state, as required:
  - If the table has been loaded using the REPLACE mode, its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables will be put in check pending no access state.
  - If the table has been loaded using the INSERT mode, its descendent immediate materialized query tables and staging tables will be put in check pending no access state, and its descendent foreign key tables will remain in their original states.
  - If the table has not been loaded, its descendent immediate materialized query tables, descendent staging tables, and its descendent foreign key tables will remain in their original states.

Examples:

Example 1: The following is an example of a query that provides information about the check pending state of tables. SUBSTR is used to extract the first 2 bytes of the CONST_CHECKED column of SYSCAT.TABLES. The first byte represents foreign key constraints, and the second byte represents check constraints. STATUS gives the check pending state, and ACCESS_MODE gives the access mode.

```
SELECT TABNAME, STATUS, ACCESS_MODE,
  SUBSTR( CONST_CHECKED, 1, 1 ) AS FK_CHECKED,
  SUBSTR( CONST_CHECKED, 2, 1 ) AS CC_CHECKED
  FROM SYSCAT.TABLES
```

Example 2: Set tables T1 and T2 to check pending no access state, and immediately cascade the check pending state to their descendants.

```
SET INTEGRITY FOR T1, T2 OFF NO ACCESS CASCADE IMMEDIATE
```

Example 3: Set parent table T1 to check pending read state without immediately cascading the check pending state to its child table T2.

```
SET INTEGRITY FOR T1 OFF READ ACCESS CASCADE DEFERRED
```

Example 4: Check integrity for T1, and get the first violation only.

```
SET INTEGRITY FOR T1 IMMEDIATE CHECKED
```

Example 5: Check integrity for T1 and T2, and put the violating rows into exception tables E1 and E2.

```
SET INTEGRITY FOR T1, T2 IMMEDIATE CHECKED
  FOR EXCEPTION IN T1 USE E1,
                   IN T2 USE E2
```

Example 6: Enable FOREIGN KEY constraint checking in T1, and CHECK constraint checking in T2, to be bypassed with the IMMEDIATE UNCHECKED option.

```
SET INTEGRITY FOR T1 FOREIGN KEY,
                                        T2 CHECK IMMEDIATE UNCHECKED
```

Example 7: Add a check constraint and a foreign key to the EMP_ACT table, using two ALTER TABLE statements. To perform constraints checking in a single pass of the table, integrity checking is turned off before the ALTER statements are invoked, and turned on after the statements are executed.

```
SET INTEGRITY FOR EMP_ACT OFF;
ALTER TABLE EMP_ACT ADD CHECK (EMSTDATE <= EMENDATE);
ALTER TABLE EMP_ACT ADD FOREIGN KEY (EMPNO) REFERENCES EMPLOYEE;
SET INTEGRITY FOR EMP_ACT IMMEDIATE CHECKED
```

Example 8: Set integrity for generated columns.

```
SET INTEGRITY FOR T1 IMMEDIATE CHECKED
  FORCE GENERATED
```

Example 9: Append (using LOAD INSERT) from different sources into an underlying table UT1 of a REFRESH IMMEDIATE materialized query table AST1, then check UT1 (incrementally) for data integrity, and refresh AST1 (incrementally). In this scenario, integrity checking for UT1 and refreshing of AST1 are incremental, because the system chooses incremental processing.

```
LOAD FROM IMTFILE1.IXF OF IXF INSERT INTO UT1;
LOAD FROM IMTFILE2.IXF OF IXF INSERT INTO UT1;
SET INTEGRITY FOR UT1 IMMEDIATE CHECKED;
REFRESH TABLE AST1;
```

Related reference:

- "Exception tables" in the *SQL Reference, Volume 1*

Related samples:

- "TbGenCol.java -- How to use generated columns (JDBC)"

SET PASSTHRU

The SET PASSTHRU statement opens and closes a session for submitting a data source's native SQL directly to that data source. The statement is not under transaction control.

Invocation:

This statement can be issued interactively. It is an executable statement that can be dynamically prepared.

Authorization:

The privileges held by the authorization ID of the statement must provide authorization to:

- Pass through to the data source.
- Satisfy security measures at the data source.

Syntax:

Description:

server-name
Names the data source for which a pass-through session is to be opened. *server-name* must identify a data source that is described in the catalog.

RESET
Closes a pass-through session.

Notes:

- The following restrictions apply to Microsoft SQL Server, Sybase, and Oracle data sources:
  - User-defined transactions cannot be used for Microsoft SQL Server and Sybase data sources in pass-through mode, because Microsoft SQL Server and Sybase restrict which SQL statements can be specified within a user-defined transaction. Because SQL statements that are processed in pass-through mode are not parsed by DataJoiner, it is not possible to detect whether the user specified an SQL statement that is permitted within a user-defined transaction.
  - The COMPUTE clause is not supported on Microsoft SQL Server and Sybase data sources.
  - DDL statements are not subject to transaction semantics on Microsoft SQL Server, Oracle and Sybase data sources. The operation, when

complete, is automatically committed by Microsoft SQL Server, Oracle or Sybase. If a rollback occurs, the DDL is not rolled back.

Examples:

Example 1: Start a pass-through session to data source BACKEND.

```
strcpy (PASS_THRU,"SET PASSTHRU BACKEND");
EXEC SQL EXECUTE IMMEDIATE :PASS_THRU;
```

Example 2: Start a pass-through session with a PREPARE statement.

```
strcpy (PASS_THRU,"SET PASSTHRU BACKEND");
EXEC SQL PREPARE STMT FROM :PASS_THRU;
EXEC SQL EXECUTE STMT;
```

Example 3: End a pass-through session.

```
strcpy (PASS_THRU_RESET,"SET PASSTHRU RESET");
EXEC SQL EXECUTE IMMEDIATE :PASS_THRU_RESET;
```

Example 4: Use the PREPARE and EXECUTE statements to end a pass-through session.

```
strcpy (PASS_THRU_RESET,"SET PASSTHRU RESET");
EXEC SQL PREPARE STMT FROM :PASS_THRU_RESET;
EXEC SQL EXECUTE STMT;
```

Example 5: Open a session to pass through to a data source, create a clustered index for a table at this data source, and close the pass-through session.

```
strcpy (PASS_THRU,"SET PASSTHRU BACKEND");
EXEC SQL EXECUTE IMMEDIATE :PASS_THRU;
EXEC SQL PREPARE STMT                 pass-through mode
  FROM "CREATE UNIQUE
        CLUSTERED INDEX TABLE_INDEX
        ON USER2.TABLE               table is not an
        WITH IGNORE DUP KEY";         alias
EXEC SQL EXECUTE STMT;
strcpy (PASS_THRU_RESET,"SET PASSTHRU RESET");
EXEC SQL EXECUTE IMMEDIATE :PASS_THRU_RESET;
```

SET PATH

The SET PATH statement changes the value of the CURRENT PATH special register. It is not under transaction control.

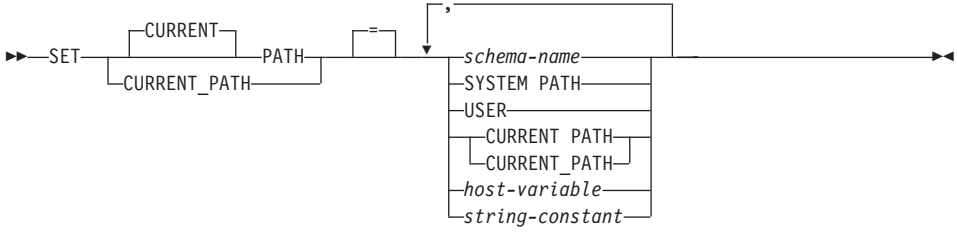
Invocation:

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
This one-part name identifies a schema that exists at the application server. No validation that the schema exists is made at the time that the path is set. If a *schema-name* is, for example, misspelled, it will not be caught, and it could affect the way subsequent SQL operates.

SYSTEM PATH
This value is the same as specifying the schema names *"SYSIBM","SYSFUN","SYSPROC"*.

USER
The value in the USER special register.

CURRENT PATH
The value of the CURRENT PATH before the execution of this statement.

host-variable
A variable of type CHAR or VARCHAR. The length of the contents of the *host-variable* must not exceed 30 bytes (SQLSTATE 42815). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

The characters of the *host-variable* must be left justified. When specifying the *schema-name* with a *host-variable*, all characters must be specified in the exact case intended as there is no conversion to uppercase characters.

string-constant
A character string constant with a maximum length of 30 bytes.

Rules:

- A schema name cannot appear more than once in the function path (SQLSTATE 42732).
- 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, doubling quotes within the schema name as necessary, and then separating each schema name by a comma. The length of the resulting string cannot exceed 254 bytes (SQLSTATE 42907).

Notes:

- Compatibilities
  - For compatibility with previous versions of DB2:
    - CURRENT FUNCTION PATH can be specified in place of CURRENT PATH
- The initial value of the CURRENT PATH special register is "SYSIBM","SYSFUN","SYSPROC","X" where X is the value of the USER special register.
- The schema SYSIBM does not need to be specified. If it is not included in the SQL path, it is implicitly assumed as the first schema (in this case, it is not included in the CURRENT PATH special register).
- The CURRENT PATH special register specifies the SQL path used to resolve user-defined data types, procedures and functions in dynamic SQL statements. The FUNCPATH bind option specifies the SQL path to be used for resolving user-defined data types and functions in static SQL statements.

Example:

Example 1: The following statement sets the CURRENT PATH special register.

```
SET PATH = FERMAT, "McDrw #8", SYSIBM
```

Example 2: The following example retrieves the current value of the CURRENT PATH special register into the host variable called CURPATH.

```
EXEC SQL VALUES (CURRENT PATH) INTO :CURPATH;
```

The value would be ″FERMAT″,″McDrw #8″,″SYSIBM″ if set by the previous example.

SET SCHEMA

The SET SCHEMA statement changes the value of the CURRENT SCHEMA special register. It is not under transaction control. If the package is bound with the DYNAMICRULES BIND option, this statement does not affect the qualifier used for unqualified database object references.

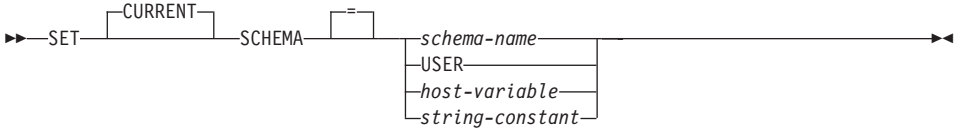
Invocation:

The 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
This one-part name identifies a schema that exists at the application server. The length must not exceed 30 bytes (SQLSTATE 42815). No validation that the schema exists is made at the time that the schema is set. If a *schema-name* is misspelled, it will not be caught, and it could affect the way subsequent SQL operates.

USER
The value in the USER special register.

host-variable
A variable of type CHAR or VARCHAR. The length of the contents of the *host-variable* must not exceed 30 (SQLSTATE 42815). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

The characters of the *host-variable* must be left justified. When specifying the *schema-name* with a *host-variable*, all characters must be specified in the exact case intended as there is no conversion to uppercase characters.

string-constant
A character string constant with a maximum length of 30.

Rules:

- If the value specified does not conform to the rules for a *schema-name*, an error (SQLSTATE 3F000) is raised.
- The value of the CURRENT SCHEMA special register is used as the schema name in all dynamic SQL statements, with the exception of the CREATE SCHEMA statement, where an unqualified reference to a database object exists.
- The QUALIFIER bind option specifies the schema name for use as the qualifier for unqualified database object names in static SQL statements.

Notes:

- The initial value of the CURRENT SCHEMA special register is equivalent to USER.
- Setting the CURRENT SCHEMA special register does not effect the CURRENT PATH special register. Hence, the CURRENT SCHEMA will not be included in the SQL path and functions, procedures and user-defined type resolution may not find these objects. To include the current schema value in the SQL path, whenever the SET SCHEMA statement is issued, also issue the SET PATH statement including the schema name from the SET SCHEMA statement.
- CURRENT SQLID is accepted as a synonym for CURRENT SCHEMA and the effect of a SET CURRENT SQLID statement will be identical to that of a SET CURRENT SCHEMA statement. No other effects, such as statement authorization changes, will occur.

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 SERVER OPTION

The SET SERVER OPTION statement specifies a server option setting that is to remain in effect while a user or application is connected to the federated database. When the connection ends, this server option's previous setting is reinstated. This statement is not under transaction control.

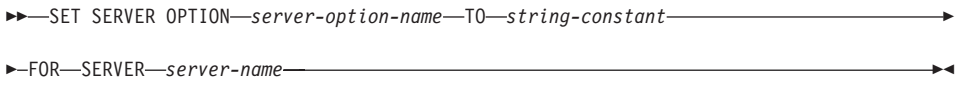
Invocation:

This statement can be issued interactively. It is an executable statement that can be dynamically prepared.

Authorization:

The authorization ID of the statement must have either SYSADM or DBADM authority on the federated database.

Syntax:



Description:

server-option-name
Names the server option that is to be set.

TO *string-constant*
Specifies the setting for *server-option-name* as a character string constant.

SERVER *server-name*
Names the data source to which *server-option-name* applies. It must be a server described in the catalog.

Notes:

- Server option names can be entered in uppercase or lowercase.
- One or more SET SERVER OPTION statements can be submitted when a user or application connects to the federated database. The statement (or statements) must be specified at the start of the first unit of work that is processed after the connection is established.
- SYSCAT.SERVEROPTIONS will not be updated based on a SET SERVER OPTION statement, because this change only affects the current connection.

Examples:

Example 1: An Oracle data source called ORASERV is defined to a federated database called DJDB. ORASERV is configured to disallow plan hints. However, the DBA would like plan hints to be enabled for a test run of a new application. When the run is over, plan hints will be disallowed again.

```
CONNECT TO DJDB;
strcpy(stmt,"set server option plan_hints to 'Y' for server oraserv");
EXEC SQL EXECUTE IMMEDIATE :stmt;
strcpy(stmt,"select c1 from ora_t1 where c1 > 100"); /*Generate plan hints*/
EXEC SQL PREPARE s1 FROM :stmt;
EXEC SQL DECLARE c1 CURSOR FOR s1;
EXEC SQL OPEN c1;
EXEC SQL FETCH c1 INTO :hv;
```

Example 2: You have set the server option PASSWORD to 'Y' (validating passwords at the data source) for all Oracle 8 data sources. However, for a particular session in which an application is connected to the federated database in order to access a specific Oracle 8 data source—one defined to the federated database DJDB as ORA8A—passwords will not need to be validated.

```
CONNECT TO DJDB;
strcpy(stmt,"set server option password to 'N' for server ora8a");
EXEC SQL PREPARE STMT_NAME FROM :stmt;
EXEC SQL EXECUTE STMT_NAME FROM :stmt;
strcpy(stmt,"select max(c1) from ora8a_t1");
EXEC SQL PREPARE STMT_NAME FROM :stmt;
EXEC SQL DECLARE c1 CURSOR FOR STMT_NAME;
EXEC SQL OPEN c1; /*Does not validate password at ora8a*/
EXEC SQL FETCH c1 INTO :hv;
```

Related reference:

- "Server options for federated systems" in the *Federated Systems Guide*

SET Variable

The SET Variable statement assigns values to local variables or to new transition variables. It is under transaction control.

Invocation:

This statement can only be used as an SQL statement in either a dynamic compound statement, trigger, SQL function or SQL method.

Authorization:

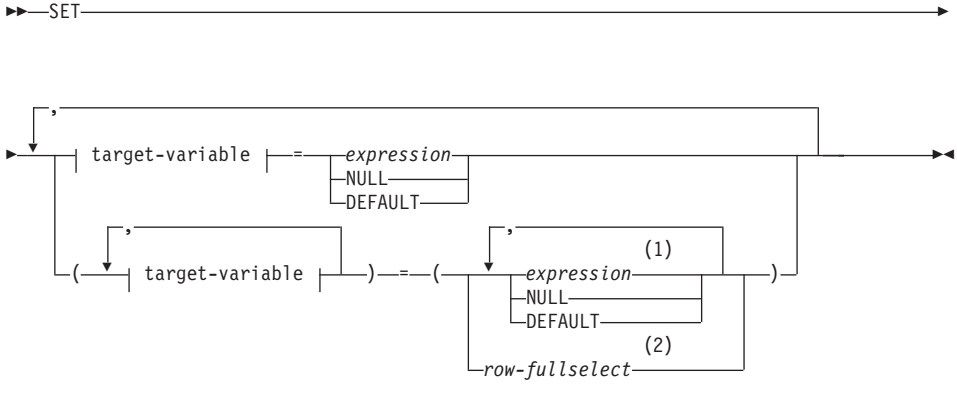
To reference a transition variable, the privileges held by the authorization ID of the trigger creator must include at least one of the following:

- UPDATE of the columns referenced on the left hand side of the assignment and SELECT for any columns referenced on the right hand side
- CONTROL privilege on the table (subject table of the trigger)
- SYSADM or DBADM authority.

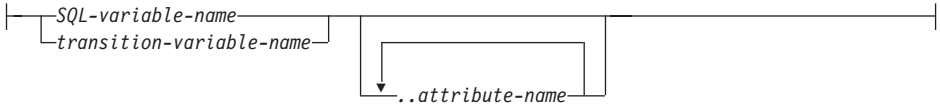
To execute this statement with a *row-fullselect* as the right hand side of the assignment, the privileges held by the authorization ID of either the trigger definer or the dynamic compound statement owner must also include at least one of the following, for each table or view referenced:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM.

Syntax:



target-variable:



Notes:

1 The number of expressions, NULLs and DEFAULTs must match the number of *target-variable*s.

2 The number of columns in the select list must match the number of *target-variable*s.

Description:

target-variable
Identifies the target variable of the assignment. A *target-variable* representing the same variable must not be specified more than once (SQLSTATE 42701).

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used. SQL variables can be defined in a dynamic compound statement.

transition-variable-name
Identifies the column to be updated in the transition row. A *transition-variable-name* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value (SQLSTATE 42703).

..attribute name
Specifies the attribute of a structured type that is set (referred to as an *attribute assignment*). The *SQL-variable-name* or *transition-variable-name* specified must be defined with a user-defined structured type (SQLSTATE 428DP). The *attribute-name* must be an attribute of the structured type (SQLSTATE 42703). An assignment that does not involve the *..attribute name* clause is referred to as a conventional assignment.

expression
Indicates the new value of the *target-variable*. The expression is any expression of the type described in "Expressions". The expression cannot include a column function except when it occurs within a scalar fullselect (SQLSTATE 42903). In the context of a CREATE TRIGGER statement, an *expression* may contain references to OLD and NEW transition variables. The transition variables must be qualified by the *correlation-name* (SQLSTATE 42702).

NULL
Specifies the null value and can only be specified for nullable columns (SQLSTATE 23502). NULL cannot be the value in an attribute assignment (SQLSTATE 429B9), unless it was specifically cast to the data type of the attribute.

DEFAULT
Specifies that the default value should be used.

If *target-variable* is a column, the value inserted depends on how the column was defined in the table.

- If the column was defined using the WITH DEFAULT clause, the value is set to the default defined for the column (see *default-clause* in "ALTER TABLE").
- If the column was defined using the IDENTITY clause, the value is generated by the database manager.
- If the column was defined without specifying the WITH DEFAULT clause, the IDENTITY clause, or the NOT NULL clause, then the value is NULL.
- If the column was defined using the NOT NULL clause and:
  - the IDENTITY clause is not used or
  - the WITH DEFAULT clause was not used or
  - DEFAULT NULL was used

  the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

If *target-variable* is an SQL variable, then the value inserted is the default as specified or implied in the variable declaration.

row-fullselect
A fullselect that returns a single row with the number of columns corresponding to the number of target-variables specified for assignment. The values are assigned to each corresponding target-variable. If the result of the row-fullselect is no rows, then null values are assigned. In the context of a CREATE TRIGGER statement, a *row-fullselect* may contain references to OLD and NEW transition variables which must be qualified by their *correlation-name* to specify which transition variable to use (SQLSTATE 42702). An error is returned if there is more than one row in the result (SQLSTATE 21000).

Rules:

- The number of values to be assigned from expressions, NULLs and DEFAULTs or the *row-fullselect* must match the number of target-variables specified for assignment (SQLSTATE 42802).

- A SET Variable statement cannot assign an SQL variable and a transition variable in one statement (SQLSTATE 42997).
- Values are assigned to target variables according to specific assignment rules.

Notes:

- If more than one assignment is included, all *expression*s and *row-fullselect*s are evaluated before the assignments are performed. Thus references to target-variables in an expression or row fullselect are always the value of the target-variable prior to any assignment in the single SET statement.
- When an identity column defined as a distinct type is updated, the entire computation is done in the source type, and the result is cast to the distinct type before the value is actually assigned to the column. (There is no casting of the previous value to the source type prior to the computation.)
- To have DB2 generate a value on a SET statement for an identity column, use the DEFAULT keyword:

  ```
  SET NEW.EMPNO = DEFAULT
  ```

  In this example, NEW.EMPNO is defined as an identity column, and the value used to update this column is generated by DB2.
- For more information on consuming values of a generated sequence for an identity column, and for information on exceeding the maximum value for an identity column, see "INSERT".

Examples:

Example 1: Set the salary column of the row for which the trigger action is currently executing to 50000.

```
SET NEW_VAR.SALARY = 50000;
```

or

```
SET (NEW_VAR.SALARY) = (50000);
```

Example 2: Set the salary and the commission column of the row for which the trigger action is currently executing to 50000 and 8000 respectively.

```
SET NEW_VAR.SALARY = 50000, NEW_VAR.COMM = 8000;
```

or

```
SET (NEW_VAR.SALARY, NEW_VAR.COMM) = (50000, 8000);
```

Example 3: Set the salary and the commission column of the row for which the trigger action is currently executing to the average of the salary and of the commission of the employees of the updated row's department respectively.

```
SET (NEW_VAR.SALARY, NEW_VAR.COMM)
   = (SELECT AVG(SALARY), AVG(COMM)
        FROM EMPLOYEE E
        WHERE E.WORKDEPT = NEW_VAR.WORKDEPT);
```

Example 4: Set the salary and the commission column of the row for which the trigger action is currently executing to 10000 and the original value of salary respectively (i.e., before the SET statement was executed).

```
SET NEW_VAR.SALARY = 10000, NEW_VAR.COMM = NEW_VAR.SALARY;
```

or

```
SET (NEW_VAR.SALARY, NEW_VAR.COMM) = (10000, NEW_VAR.SALARY);
```

Related reference:

- "Expressions" in the *SQL Reference, Volume 1*
- "ALTER TABLE" on page 41
- "INSERT" on page 603
- "Assignments and comparisons" in the *SQL Reference, Volume 1*

UPDATE

The UPDATE statement updates the values of specified columns in rows of a table, view or nickname, or executes the INSTEAD OF update trigger. Updating a row of a view updates a row of its base table, if no INSTEAD OF trigger is defined for the update operation on this view. If such a trigger is defined, the trigger will be executed instead. Updating a row using a nickname updates a row in the data source object to which the nickname refers.

The forms of this statement are:

- 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:

An UPDATE statement can be embedded in an application program or issued through the use of dynamic SQL statements. 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:

- UPDATE privilege on the table, view or nickname where rows are to be updated
- UPDATE privilege on each of the columns to be updated.
- CONTROL privilege on the table, view or nickname where rows are to be updated
- SYSADM or DBADM authority.
- If a *row-fullselect* is included in the assignment, at least one of the following for each referenced table, view or nickname:
  - SELECT privilege
  - CONTROL privilege
  - SYSADM or DBADM authority.

For each table, view or nickname referenced by a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SELECT privilege
- CONTROL privilege

- SYSADM or DBADM authority.

If the package used to process the statement is precompiled with SQL92 rules (option LANGLEVEL with a value of SQL92E or MIA), and the searched form of an UPDATE statement includes a reference to a column of the table, view or nickname in the right side of the *assignment-clause*, or anywhere in the *search-condition*, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

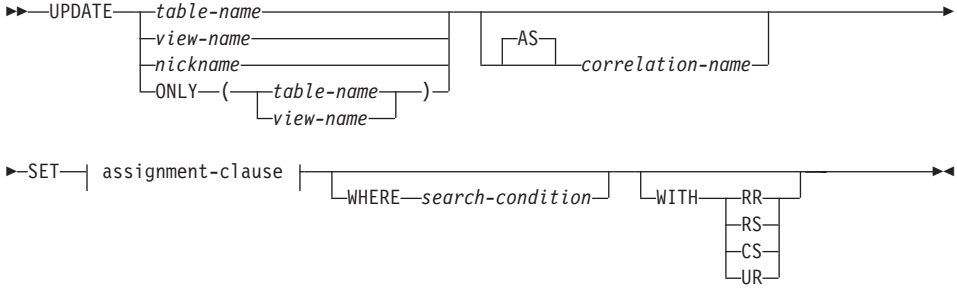
If the specified table or view is preceded by the ONLY keyword, the privileges held by the authorization ID of the statement must also include the SELECT privilege for every subtable or subview of the specified table or view.

GROUP privileges are not checked for static UPDATE statements.

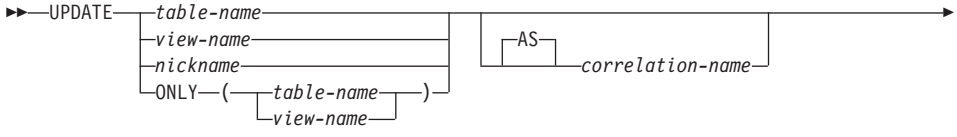
If the target of the update operation is a nickname, the privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges required for the operation on the object at the data source. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

Syntax:

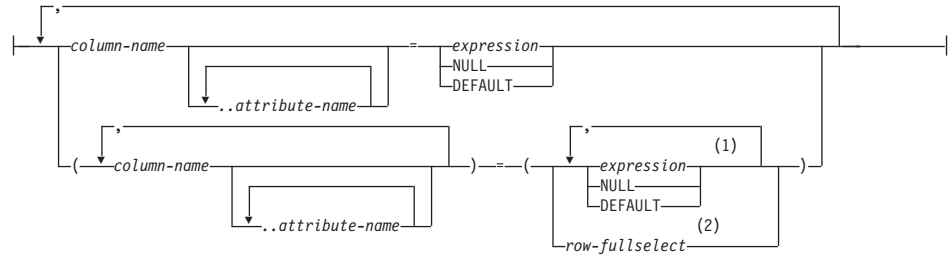
Searched UPDATE:



Positioned UPDATE:



assignment-clause:



Notes:

1 The number of expressions, NULLs and DEFAULTs must match the number of *column-name*s.

2 The number of columns in the select list must match the number of *column-name*s.

Description:

table-name, *view-name* **or** *nickname*
Is the name of the table, view or nickname to be updated. The name must identify a table, view or nickname described in the catalog, but not a catalog table, a view of a catalog table (unless it is one of the updatable SYSSTAT views), a system-maintained materialized query table, or a read-only view that has no INSTEAD OF trigger defined for its update operations.

If *table-name* is a typed table, rows of the table or any of its proper subtables may get updated by the statement. Only the columns of the specified table may be set or referenced in the WHERE clause. For a positioned UPDATE, the associated cursor must also have specified the same table, view or nickname in the FROM clause without using ONLY.

ONLY (*table-name***)**
Applicable to typed tables, the ONLY keyword specifies that the statement should apply only to data of the specified table and rows of proper subtables cannot be updated by the statement. For a positioned UPDATE, the associated cursor must also have specified the table in the FROM clause using ONLY. If *table-name* is not a typed table, the ONLY keyword has no effect on the statement.

ONLY (*view-name***)**
Applicable to typed views, the ONLY keyword specifies that the statement should apply only to data of the specified view and rows of proper subviews cannot be updated by the statement. For a positioned UPDATE,

the associated cursor must also have specified the view in the FROM clause using ONLY. If *view-name* is not a typed view, the ONLY keyword has no effect on the statement.

AS
Optional keyword to introduce the *correlation-name*.

correlation-name
May be used within *search-condition* or *assignment-clause* to designate the table, view or nickname. For an explanation of *correlation-name*, see "Identifiers".

SET
Introduces the assignment of values to column names.

assignment-clause

column-name
Identifies a column to be updated. The *column-name* must identify an updatable column of the specified table, view or nickname. The object ID column of a typed table is not updatable (SQLSTATE 428DZ). A column must not be specified more than once, unless it is followed by *..attribute-name* (SQLSTATE 42701).

For a Positioned UPDATE:

- If the *update-clause* was specified in the *select-statement* of the cursor, each column name in the *assignment-clause* must also appear in the *update-clause*.
- If the *update-clause* was not specified in the *select-statement* of the cursor and LANGLEVEL MIA or SQL92E was specified when the application was precompiled, the name of any updatable column may be specified.
- If the *update-clause* was not specified in the *select-statement* of the cursor and LANGLEVEL SAA1 was specified either explicitly or by default when the application was precompiled, no columns may be updated.

..attribute-name
Specifies the attribute of a structured type that is set (referred to as an *attribute assignment*. The *column-name* specified must be defined with a user-defined structured type (SQLSTATE 428DP). The attribute-name must be an attribute of the structured type of *column-name* (SQLSTATE 42703). An assignment that does not involve the *..attribute-name* clause is referred to as a *conventional assignment*.

expression
Indicates the new value of the column. The expression is any

expression of the type described in "Expressions". The expression cannot include a column function except when it occurs within a scalar fullselect (SQLSTATE 42903).

An *expression* 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.

NULL
Specifies the null value and can only be specified for nullable columns (SQLSTATE 23502). NULL cannot be the value in an attribute assignment (SQLSTATE 429B9) unless it is specifically cast to the data type of the attribute.

DEFAULT
Specifies that the default value should be used based on how the corresponding column is defined in the table. The value that is inserted depends on how the column was defined.

- If the column was defined as a generated column based on an expression, the column value will be generated by the system, based on the expression.
- If the column was defined using the IDENTITY clause, the value is generated by the database manager.
- If the column was defined using the WITH DEFAULT clause, the value is set to the default defined for the column (see *default-clause* in "ALTER TABLE").
- If the target of the update operation is an INSTEAD OF trigger, or if the column was defined without specifying the WITH DEFAULT clause, the GENERATED clause, or the NOT NULL clause, then the value used is NULL.
- If the column was defined using the NOT NULL clause and the GENERATED clause was not used, or the WITH DEFAULT clause was not used, or DEFAULT NULL was used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

The only value that a generated column defined with the GENERATED ALWAYS clause can be set to is DEFAULT (SQLSTATE 428C9).

The DEFAULT keyword cannot be used as the value in an attribute assignment (SQLSTATE 429B9).

The DEFAULT keyword cannot be used as the value in an assignment for update on a nickname where the data source does not support DEFAULT syntax.

row-fullselect
A fullselect that returns a single row with the number of columns corresponding to the number of *column-name*s specified for assignment. The values are assigned to each corresponding *column-name*. If the result of the *row-fullselect* is no rows, then null values are assigned.

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. An error is returned if there is more than one row in the result (SQLSTATE 21000).

WHERE
Introduces a condition that indicates what rows are updated. You can omit the clause, give a search condition, or name a cursor. If the clause is omitted, all rows of the table, view or nickname are updated.

search-condition
Each *column-name* in the search condition, other than in a subquery, must name a column of the table, view or nickname. When the search condition includes a subquery in which the same table is the base object of both the UPDATE and the subquery, the subquery is completely evaluated before any rows are updated.

The search-condition is applied to each row of the table, view or nickname and the updated rows are those for which the result of the search-condition is true.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, a subquery with no correlated references is 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, explained in "DECLARE CURSOR". The DECLARE CURSOR statement must precede the UPDATE statement in the program.

The table, view or nickname named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR".)

When the UPDATE statement is executed, the cursor must be positioned on a row; that row is updated.

This form of UPDATE cannot be used if the target of the update is a view on which an INSTEAD OF UPDATE trigger is defined, or a view that includes an OLAP function in the select list of the fullselect that defines the view (SQLSTATE 42828).

WITH
Specifies the isolation level at which the UPDATE statement is executed.

RR
Repeatable Read

RS
Read Stability

CS
Cursor Stability

UR
Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound.

Rules:

- ***Triggers:*** UPDATE statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the update values. If an update operation on a view causes an INSTEAD OF trigger to fire, validity, referential integrity, and constraints will be checked against the updates that are performed in the trigger, and not against the view that caused the trigger to fire, or its underlying tables.
- ***Assignment***: Update values are assigned to columns according to specific assignment rules.
- ***Validity***: The updated row must conform to any constraints imposed on the table (or on the base table of the view) by any unique index on an updated column.

  If a view is used that is not defined using WITH CHECK OPTION, rows can be changed so that they no longer conform to the definition of the view. Such rows are updated in the base table of the view and no longer appear in the view.

  If a view is used that is defined using WITH CHECK OPTION, an updated row must conform to the definition of the view. For an explanation of the rules governing this situation, see "CREATE VIEW".
- ***Check Constraint***: Update value must satisfy the check-conditions of the check constraints defined on the table.

  An UPDATE to a table with check constraints defined has the constraint conditions for each column updated evaluated once for each row that is

updated. When processing an UPDATE statement, only the check constraints referring to the updated columns are checked.

- ***Referential Integrity***: The value of the parent unique keys cannot be changed if the update rule is RESTRICT and there are one or more dependent rows. However, if the update rule is NO ACTION, parent unique keys can be updated as long as every child has a parent key by the time the update statement completes. A non-null update value of a foreign key must be equal to a value of the primary key of the parent table of the relationship.

Notes:

- If an update value violates any constraints, or if any other error occurs during the execution of the UPDATE statement, no rows are updated. The order in which multiple rows are updated is undefined.
- When an UPDATE statement completes execution, the value of SQLERRD(3) in the SQLCA is 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. The SQLERRD(5) field contains the number of rows inserted, deleted, or updated by all activated triggers.
- Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful UPDATE statement. Until the locks are released, the updated row can only be accessed by the application process that performed the update (except for applications using the Uncommitted Read isolation level). For further information on locking, see the descriptions of the COMMIT, ROLLBACK, and LOCK TABLE statements.
- If the URL value of a DATALINK column is updated, this is the same as deleting the old DATALINK value then inserting the new one. First, if the old value was linked to a file, that file is unlinked. Then, unless the linkage attributes of the DATALINK value are empty, the specified file is linked to that column. The only exception to this is that if the URL of the new DATALINK value is *identical* to the URL of the existing DATALINK value, there is no need to communicate with the associated Data Links Manager to unlink and relink the same file. In this situation, the overhead is entirely eliminated.

  The comment value of a DATALINK column can be updated without relinking the file by specifying an empty string as the URL path (for example, as the *data-location* argument of the DLVALUE scalar function or by specifying the new value to be the same as the old value).

  If a DATALINK column is updated with a null, it is the same as deleting the existing DATALINK value.

An error may occur when attempting to update a DATALINK value if the file server of either the existing value or the new value is no longer registered with the database server (SQLSTATE 55022).

- When updating the column distribution statistics for a typed table, the subtable that first introduced the column must be specified.
- Multiple attribute assignments on the same structured type column occur in the order specified in the SET clause and, within a parenthesized set clause, in left-to-right order.
- An attribute assignment invokes the mutator method for the attribute of the user-defined structured type. For example, the assignment `st..a1=x` has the same effect as using the mutator method in the assignment `st = st..a1(x)`.
- While a given column may be a target column in only one conventional assignment, a column may be a target column in multiple attribute assignments (but only if it is not also a target column in a conventional assignment).
- When an identity column defined as a distinct type is updated, the entire computation is done in the source type, and the result is cast to the distinct type before the value is actually assigned to the column. (There is no casting of the previous value to the source type prior to the computation.)
- To have DB2 generate a value on a SET statement for an identity column, use the DEFAULT keyword:

  ```
  SET NEW.EMPNO = DEFAULT
  ```

  In this example, NEW.EMPNO is defined as an identity column, and the value used to update this column is generated by DB2.
- For more information about consuming values of a generated sequence for an identity column, or about exceeding the maximum value for an identity column, see "INSERT".

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'
```

This statement could also be written as follows.

```
UPDATE EMPLOYEE
  SET (JOB, SALARY, BONUS, COMM) = (NULL, 0, 0, 0)
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'
```

- *Example 4:* Update the salary and the commission column of the employee with employee number 000120 to the average of the salary and of the commission of the employees of the updated row's department respectively.

```
UPDATE EMPLOYEE EU
  SET (EU.SALARY, EU.COMM)
  =
(SELECT AVG(ES.SALARY), AVG(ES.COMM)
  FROM EMPLOYEE ES
  WHERE ES.WORKDEPT = EU.WORKDEPT)
  WHERE EU.EMPNO = '000120'
```

- *Example 5:* In a C program display the rows from the EMPLOYEE table and then, if requested to do so, change the job (JOB) of certain employees to the new job keyed in.

```
EXEC SQL  DECLARE C1 CURSOR FOR
               SELECT *
                 FROM EMPLOYEE
                 FOR UPDATE OF JOB;

EXEC SQL  OPEN C1;

EXEC SQL  FETCH C1 INTO ...     ;
if ( strcmp (change, "YES") == 0 )
  EXEC SQL  UPDATE EMPLOYEE
              SET JOB = :newjob
              WHERE CURRENT OF C1;

EXEC SQL  CLOSE C1;
```

- *Example 6:* These examples mutate attributes of column objects.

  Assume that the following types and tables exist:

```
CREATE TYPE POINT AS (X INTEGER, Y INTEGER)
  NOT FINAL WITHOUT COMPARISONS
  MODE DB2SQL

CREATE TYPE CIRCLE AS (RADIUS INTEGER, CENTER POINT)
  NOT FINAL WITHOUT COMPARISONS
  MODE DB2SQL

CREATE TABLE CIRCLES (ID INTEGER, OWNER VARCHAR(50), C CIRCLE
```

  The following example updates the CIRCLES table by changing the OWNER column and the RADIUS attribute of the CIRCLE column where the ID is 999:

```
UPDATE CIRCLES
  SET OWNER = 'Bruce'
    C..RADIUS = 5
  WHERE ID = 999
```

The following example transposes the X and Y coordinates of the center of the circle identified by 999:

```
UPDATE CIRCLES
  SET C..CENTER..X = C..CENTER..Y,
    C..CENTER..Y = C..CENTER..X
  WHERE ID = 999
```

The following example is another way of writing both of the above statements. This example combines the effects of both of the above examples:

```
UPDATE CIRCLES
  SET (OWNER,C..RADIUS,C..CENTER..X,C..CENTER..Y) =
    ('Bruce',5,C..CENTER..Y,C..CENTER..X)
  WHERE ID = 999
```

Related reference:

- "Identifiers" in the *SQL Reference, Volume 1*
- "Expressions" in the *SQL Reference, Volume 1*
- "Search conditions" in the *SQL Reference, Volume 1*
- "ALTER TABLE" on page 41
- "CREATE VIEW" on page 463
- "DECLARE CURSOR" on page 482
- "INSERT" on page 603
- "SQLCA (SQL communications area)" in the *SQL Reference, Volume 1*
- "Assignments and comparisons" in the *SQL Reference, Volume 1*

Related samples:

- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"
- "spserver.sqc -- A variety of types of stored procedures (C)"
- "tbmod.sqc -- How to modify table data (C)"
- "tut_mod.sqc -- How to modify table data (C)"
- "dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)"
- "spserver.sqC -- A variety of types of stored procedures (C++)"
- "tbmod.sqC -- How to modify table data (C++)"
- "tut_mod.sqC -- How to modify table data (C++)"
- "SpServer.java -- Provide a variety of types of stored procedures to be called from (JDBC)"

- "TbMod.java -- How to modify table data (JDBC)"
- "TutMod.java -- Modify data in a table (JDBC)"
- "SpServer.sqlj -- Provide a variety of types of stored procedures to be called from (SQLj)"
- "TbMod.sqlj -- How to modify table data (SQLj)"
- "TutMod.sqlj -- Modify data in a table (SQLj)"
- "tbmod.c -- How to modify table data (CLI)"
- "tut_mod.c -- How to modify table data (CLI)"
- "updat.sqb -- How to update, delete and insert table data (MF COBOL)"
- "varinp.sqb -- How to update table data using parameter markers (MF COBOL)"

VALUES

The VALUES statement is a form of query. It can be embedded in an application program or issued interactively.

Related reference:

- "Fullselect" in the *SQL Reference, Volume 1*

Related samples:

- "dtlob.c -- How to read and write LOB data (CLI)"
- "dtlob.sqc -- How to use the LOB data type (C)"
- "fnuse.sqc -- How to use built-in SQL functions (C)"
- "spserver.sqc -- A variety of types of stored procedures (C)"
- "dtlob.sqC -- How to use the LOB data type (C++)"
- "fnuse.sqC -- How to use built-in SQL functions (C++)"
- "spserver.sqC -- A variety of types of stored procedures (C++)"
- "lobloc.sqb -- Demonstrates the use of LOB locators (MF COBOL)"

VALUES INTO

The VALUES INTO statement produces a result table consisting of at most one row and assigns the values in that row to host variables.

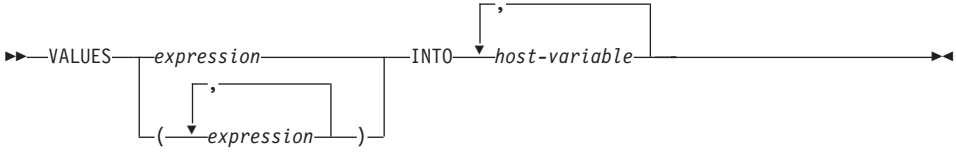
Invocation:

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization:

None required.

Syntax:



Description:

VALUES
Introduces a single row consisting of one of more columns.

expression
An expression that defines a single value of a one column result table.

(expression,...)
One or more expressions that define the values for one or more columns of the result table.

INTO
Introduces a list of host variables.

host-variable
Identifies a variable that is described in the program under the rules for declaring host 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. If the number of host variables is less than the number of column values, the value 'W' is assigned to the SQLWARN3 field of the SQLCA.

Each assignment to a variable is made in sequence through the list. If an error occurs, no value is assigned to any host variable.

Examples:

Example 1: This C example retrieves the value of the CURRENT PATH special register into a host variable.

```
EXEC SQL VALUES(CURRENT PATH)
         INTO :hvl;
```

Example 2: This C example retrieves a portion of a LOB field into a host variable, exploiting the LOB locator for deferred retrieval.

```
EXEC SQL VALUES (substr(:locator1,35))
         INTO :details;
```

Related reference:

- "SQLCA (SQL communications area)" in the *SQL Reference, Volume 1*
- "Assignments and comparisons" in the *SQL Reference, Volume 1*

WHENEVER

The WHENEVER statement specifies the action to be taken when a specified exception condition occurs.

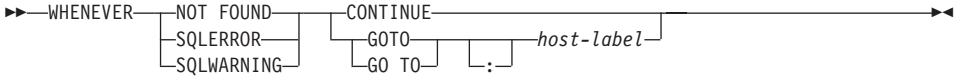
Invocation:

This statement can only be embedded in an application program. It is not an executable statement. The statement is not supported in REXX.

Authorization:

None required.

Syntax:



Description:
The NOT FOUND, SQLERROR, or SQLWARNING clause is used to identify the type of exception condition.

NOT FOUND
Identifies any condition that results in an SQLCODE of +100 or an SQLSTATE of '02000'.

SQLERROR
Identifies any condition that results in a negative SQLCODE.

SQLWARNING
Identifies any condition that results in a warning condition (SQLWARN0 is 'W'), or that results in a positive SQL return code other than +100.

The CONTINUE or GO TO clause is used to specify what is to happen when the identified type of exception condition exists.

CONTINUE
Causes the next sequential instruction of the source program to be executed.

GOTO or **GO TO** *host-label*
Causes control to pass to 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.

Notes:

There are three types of WHENEVER statements:

- WHENEVER NOT FOUND
- WHENEVER SQLERROR
- WHENEVER SQLWARNING

Every executable SQL statement in a program is within the scope of one implicit or explicit WHENEVER statement of each type. The scope of a WHENEVER statement is related to the listing sequence of the statements in the program, not their execution sequence.

An SQL statement is within the scope of the last WHENEVER statement of each type that is specified before that SQL statement in the source program. If a WHENEVER statement of some type is not specified before an SQL statement, that SQL statement is within the scope of an implicit WHENEVER statement of that type in which CONTINUE is specified.

Example:

In the following C example, if an error is produced, go to HANDLERR. If a warning code is produced, continue with the normal flow of the program. If no data is returned, go to ENDDATA.

```
EXEC SQL WHENEVER SQLERROR GOTO HANDLERR;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER NOT FOUND GO TO ENDDATA;
```

Related samples:

- "outsrv.sqb -- Demonstrates stored procedures using the SQLDA structure (MF COBOL)"
- "spserver.sqc -- A variety of types of stored procedures (C)"
- "spserver.sqC -- A variety of types of stored procedures (C++)"

Chapter 2. SQL control statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the statements that constitute the body of an SQL routine, trigger, or dynamic compound statement.

About SQL control statements

Control statements are SQL statements that allow structured query language 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 statements can be used in the body of a routine, trigger or a dynamic compound statement.

SQL parameters and SQL variables can be referenced anywhere in the statement where an expression or host 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, and can be qualified with the label name specified at the beginning of the compound statement.

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. In this case, the name should be explicitly qualified to indicate whether it is a column, SQL variable, or SQL parameter.

If the name is not qualified, the following rules describe whether the name refers to the column or to the SQL variable or parameter:

- If the tables and views specified in an SQL routine body exist at the time the routine is created, the name is first checked as a column name. If not found as a column, it is then checked as an SQL variable or SQL parameter name.
- If the referenced tables or views do not exist at the time the routine is created, the name is first checked as an SQL variable or SQL parameter name. If not found, it is assumed to be a column.

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:

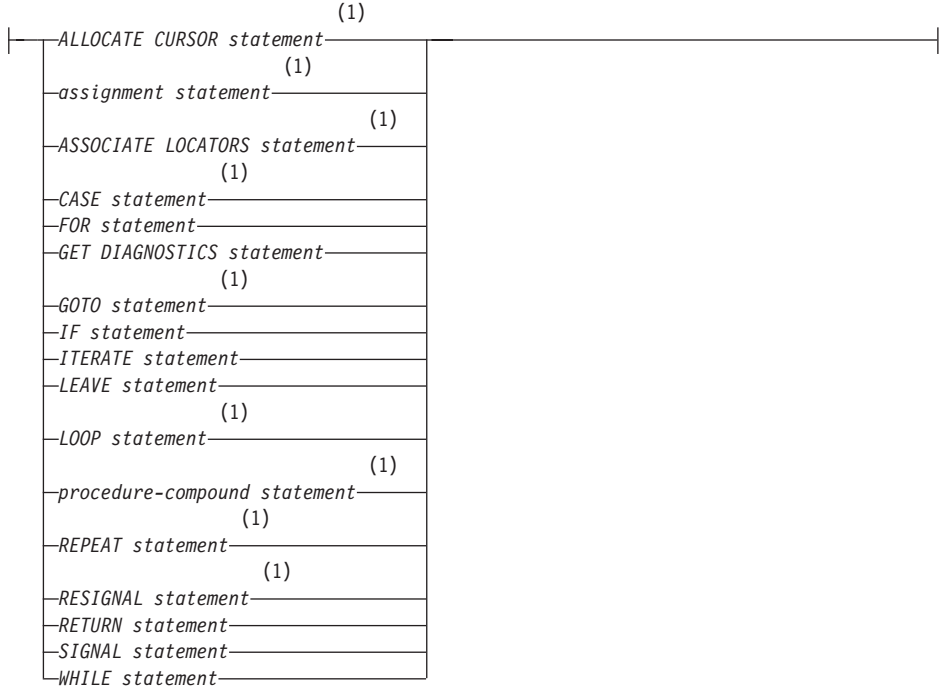
- In the SET PATH and SET SCHEMA statements, the name is checked as an SQL parameter or SQL variable name. If not found as an SQL variable or SQL parameter name, it is used as an identifier.
- In the CONNECT statement, the name is used as an identifier.

SQL procedure statement

An SQL Procedure statement is an SQL control statement or an SQL statement that is specified within a control statement of an SQL routine body.

Syntax:

SQL-control-statement:



Notes:

1 This statement is only supported in the scope of an SQL Procedure.

Description:

label:

Specifies the label for an SQL procedure statement. The label must be unique within a list of SQL procedure statements, including any compound statements nested within the list. Note that compound statements that are not nested may use the same label. A list of SQL procedure statements is possible in a number of SQL control statements.

In the context of a trigger, SQL function, SQL method, or dynamic compound statement, only the FOR statement, WHILE statement, and the dynamic compound statement may include a label.

SQL-statement

In the body of an SQL procedure, all executable SQL statements can be contained, with the exception of the following:

- CONNECT
- CREATE any object other than indexes, tables, or views
- DESCRIBE
- DISCONNECT
- DROP any object other than indexes, tables, or views
- FLUSH EVENT MONITOR
- REFRESH TABLE
- RELEASE (connection only)
- RENAME TABLE
- RENAME TABLESPACE
- REVOKE
- SET CONNECTION
- SET INTEGRITY

For SQL statements that are allowed in these contexts, see "Compound SQL (Dynamic)" and "CREATE TRIGGER".

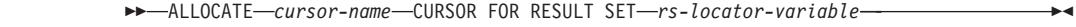
Related reference:

- "CREATE TRIGGER" on page 414
- "Compound SQL (Dynamic)" on page 123

ALLOCATE CURSOR statement

The ALLOCATE CURSOR statement allocates a cursor for the result set identified by the result set locator variable. For more information on result set locator variables, see "ASSOCIATE LOCATORS statement".

Syntax:



Description:

cursor-name
Names the cursor. The name must not identify a cursor that has already been declared in the source SQL procedure (SQLSTATE 24502).

CURSOR FOR RESULT SET *rs-locator-variable*

Names a result set locator variable that has been declared in the source SQL procedure, according to the rules for host variables. For more information on declaring SQL variables, see "Compound statement (Procedure)".

The result set locator variable must contain a valid result set locator value, as returned by the ASSOCIATE LOCATORS SQL statement (SQLSTATE 24501).

Rules:

- The following rules apply when using an allocated cursor:
  - An allocated cursor cannot be opened with the OPEN statement (SQLSTATE 24502).
  - An allocated cursor cannot be used in a positioned UPDATE or DELETE statement (SQLSTATE 42828).
  - An allocated cursor can be closed with the CLOSE statement. Closing an allocated cursor closes the associated cursor.
  - Only one cursor can be allocated to each result set.
- Allocated cursors last until a rollback operation, an implicit close, or an explicit close.
- A commit operation destroys allocated cursors that are not defined WITH HOLD.
- Destroying an allocated cursor closes the associated cursor in the SQL procedure.

Examples:

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;
```

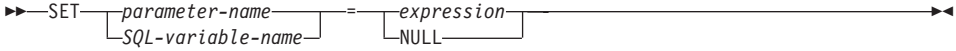
Related reference:

- "Compound statement (Procedure)" on page 767
- "ASSOCIATE LOCATORS statement" on page 762

Assignment statement

The assignment statement assigns a value to an output parameter, a local variable, or a special register.

Syntax:



Description:

parameter-name
Identifies the parameter that is the assignment target. The parameter must be specified in *parameter-declaration* in the CREATE PROCEDURE statement and must be defined as an OUT or INOUT parameter.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used. SQL variables can be defined in a compound statement.

expression **or NULL**
Specifies the expression or value that is the source for the assignment.

Rules:

- Assignment statements in SQL procedures must conform to the SQL assignment rules. String assignments use storage assignment rules.
- If a variable has been declared with an identifier that matches the name of a special register (such as PATH), then the variable must be delimited to distinguish it from assignment to the special register (for example, SET "PATH" = 1; for a variable called PATH declared as an integer).

Examples:

Increase the SQL variable `p_salary` by 10 percent.

```
SET p_salary = p_salary + (p_salary * .10)
```

Set SQL variable `p_salary` to the null value.

```
SET p_salary = NULL
```

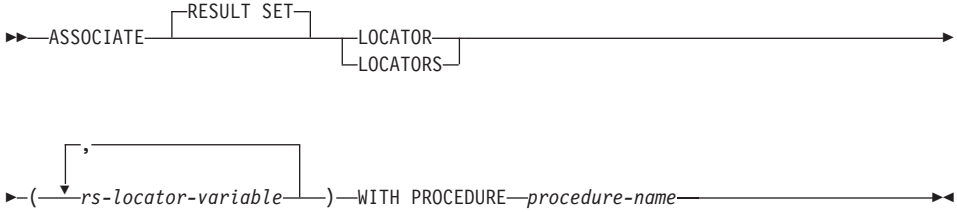
Related reference:

- "Assignments and comparisons" in the *SQL Reference, Volume 1*

ASSOCIATE LOCATORS statement

The ASSOCIATE LOCATORS statement gets the result set locator value for each result set returned by a stored procedure.

Syntax:



Description:

rs-locator-variable
Specifies a result set locator variable that has been declared in a compound statement.

WITH PROCEDURE
Identifies the stored procedure that returns result set locators by the specified procedure name.

procedure-name
A procedure name is a qualified or unqualified name. Each part of the name must be composed of SBCS characters.

A fully qualified procedure name is a two-part name. The first part is an identifier that contains the schema name of the stored procedure. The last part is an identifier that contains the name of the stored procedure. A period must separate each of the parts. Any or all of the parts can be a delimited identifier.

If the procedure name is unqualified, it has only one name because the implicit schema name is not added as a qualifier to the procedure name. Successful execution of the ASSOCIATE LOCATOR statement only requires that the unqualified procedure name in the statement is the same as the procedure name in the most recently executed CALL statement that was specified with an unqualified procedure name. The implicit schema name for the unqualified name in the CALL statement is not considered in the match. The rules for how the procedure name must be specified are described below.

When the ASSOCIATE LOCATORS statement is executed, the procedure name or specification must identify a stored procedure that the requester has already invoked using the 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 name was specified on the CALL statement, you must use a two-part name in the ASSOCIATE LOCATORS statement.

Rules:

- More than one locator can be assigned to a result set. You can issue the same ASSOCIATE LOCATORS statement more than once with different result set locator variables.
- If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is less than the number of locators returned by the stored procedure, all variables in the statement are assigned a value, and a warning is issued.
- 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 stored procedure, the extra variables are assigned a value of 0.
- If a stored procedure is called more than once from the same caller, only the most recent result sets are accessible.

Examples:

The statements in the following examples are assumed to be embedded in SQL Procedures.

Example 1: Use result set locator variables LOC1 and LOC2 to get the result set locator values for the two result sets returned by stored procedure P1. Assume that the stored procedure is called with a one-part name.

```
CALL P1;
ASSOCIATE RESULT SET LOCATORS (LOC1, LOC2)
  WITH PROCEDURE P1;
```

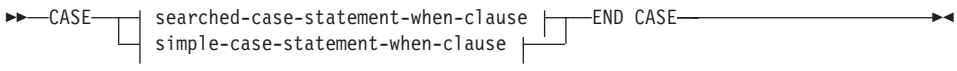
Example 2: Repeat the scenario in Example 1, but use a two-part name to specify an explicit schema name for the stored procedure to ensure that stored procedure P1 in schema MYSCHEMA is used.

```
CALL MYSCHEMA.P1;
ASSOCIATE RESULT SET LOCATORS (LOC1, LOC2)
  WITH PROCEDURE MYSCHEMA.P1;
```

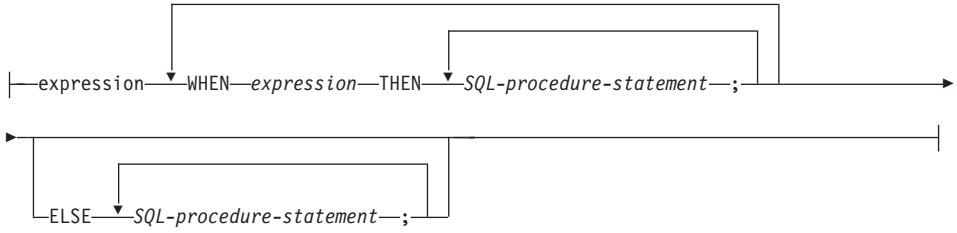
CASE statement

The CASE statement selects an execution path based on multiple conditions.

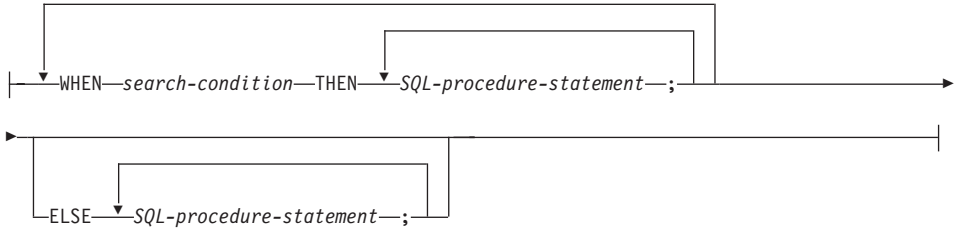
Syntax:



simple-case-statement-when-clause:



searched-case-statement-when-clause:



Description:

CASE
Begins a *case-expression*.

simple-case-statement-when-clause
The value of the *expression* prior to the first WHEN keyword is tested for equality with the value of each *expression* that follows the WHEN keyword. If the search condition is true, the THEN statement is executed. If the result is unknown or false, processing continues to the next search

condition. If the result does not match any of the search conditions, and an ELSE clause is present, the statements in the ELSE clause are processed.

searched-case-statement-when-clause
The *search-condition* following the WHEN keyword is evaluated. If it evaluates to true, the statements in the associated THEN clause are processed. 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 processed.

SQL-procedure-statement
Specifies a statement that should be invoked.

END CASE
Ends a *case-statement*.

Notes:

- If none of the conditions specified in the WHEN are true, and an ELSE clause is not specified, an error is issued at runtime, and the execution of the case statement is terminated (SQLSTATE 20000).
- Ensure that your CASE statement covers all possible execution conditions.

Examples:

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-case-statement-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
```

The following example shows how to do this using the syntax for a *searched-case-statement-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
```

Related samples:

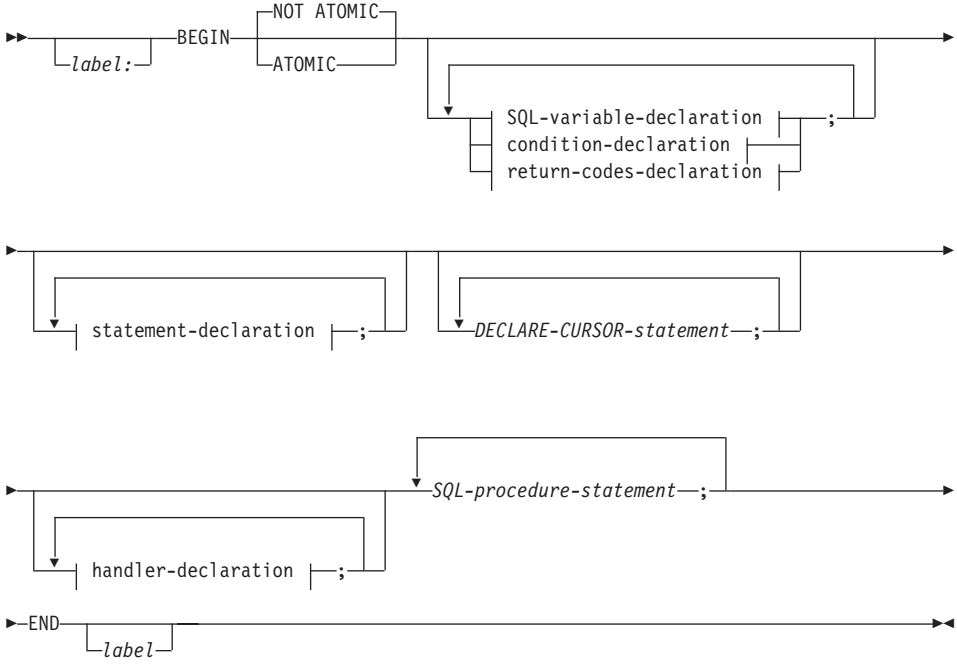
- "advsql.sqb -- How to read table data using CASE (MF COBOL)"
- "tbtrig.sqc -- How to use a trigger on a table (C)"
- "tbtrig.sqC -- How to use a trigger on a table (C++)"
- "TbTrig.java -- How to use triggers (JDBC)"
- "TbTrig.sqlj -- How to use triggers (SQLj)"

Compound statement (Procedure)

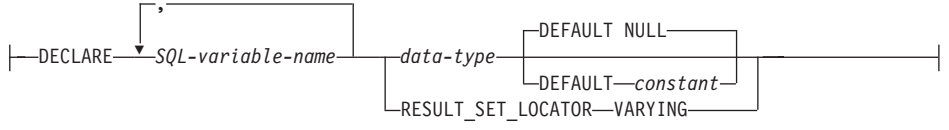
A procedure compound statement groups other statements together in an SQL procedure. You can declare SQL variables, cursors, and condition handlers within a compound statement.

Syntax:

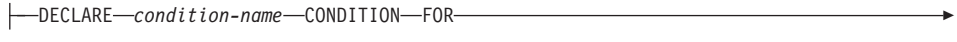
procedure-compound-statement

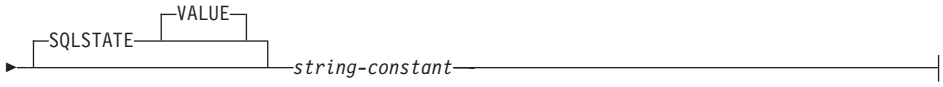


SQL-variable-declaration:

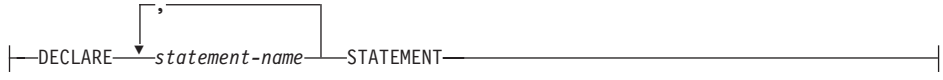


condition-declaration:

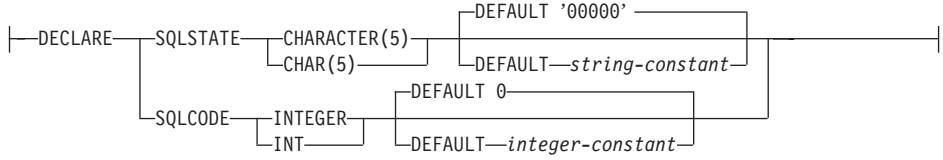




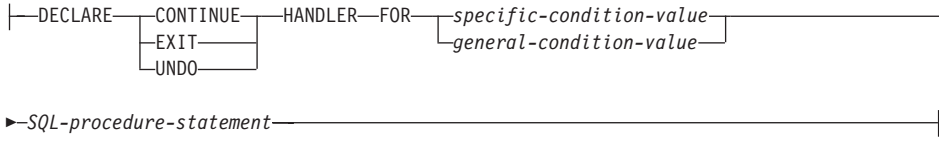
statement-declaration:



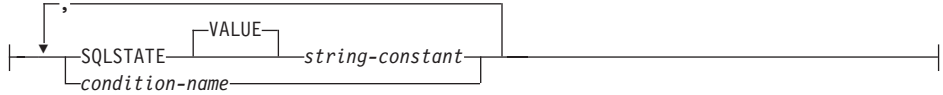
return-codes-declaration:



handler-declaration:



specific-condition-value:



general-condition-value:

Description:

label

Defines the label for the code block. If the beginning label is specified, it can be used to qualify SQL variables declared in the compound statement and can also be specified on a LEAVE statement. If the ending label is specified, it must be the same as the beginning label.

ATOMIC or NOT ATOMIC
ATOMIC indicates that if an unhandled exception condition occurs in the compound statement, all SQL statements in the compound statement will be rolled back. 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 a variable that is local to the compound statement.

SQL-variable-name
Defines the name of a local variable. DB2 converts all SQL variable names to uppercase. The name cannot be the same as another SQL variable within the same compound statement and cannot be the same as a parameter name. SQL variable names should not be the same as column names. If an SQL statement contains an identifier with the same name as an SQL variable and a column reference, DB2 interprets the identifier as a column. If the compound statement where the variable is declared is labeled, then uses of the variable can be qualified with the label. For example, variable V declared in compound statement labeled C can be referred to as C.V.

data-type
Specifies the data type of the variable. User-defined data types, graphic types, and FOR BIT DATA are not supported.

DEFAULT *constant* **or NULL**
Defines the default for the SQL variable. The variable is initialized when the SQL procedure is called. If a default value is not specified, the variable is initialized to NULL.

RESULT_SET_LOCATOR VARYING
Specifies the data type for a result set locator variable.

condition-declaration
Declares a condition name and corresponding SQLSTATE value.

condition-name
Specifies the name of the condition. The condition name must be unique within the procedure body and can be referenced only 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 enclosed in single quotes, and cannot be '00000'.

statement-declaration
Declares a list of one or more names that are local to the compound

statement. A statement name cannot be the same as another statement name within the same compound statement.

return-codes-declaration
Declares special variables called SQLSTATE and SQLCODE that are set automatically to the value returned after processing an SQL statement. Both the SQLSTATE and SQLCODE variables can only be declared in the outermost compound statement of the SQL procedure body. These variables may be declared only once per SQL procedure.

declare-cursor-statement
Declares a cursor in the procedure body. Each cursor must have a unique name. The cursor can be referenced only from within the compound statement. Use an OPEN statement to open the cursor, and a FETCH statement to read rows using the cursor. To return result sets from the SQL procedure to the client application, the cursor must be declared using the WITH RETURN clause. The following example returns one result set to the client application:

```
CREATE PROCEDURE RESULT_SET()
  LANGUAGE SQL
  RESULT SETS 1
  BEGIN
    DECLARE C1 CURSOR WITH RETURN FOR
      SELECT id, name, dept, job
        FROM staff;
    OPEN C1;
END
```

Note: To process result sets, you must write your client application using one of the DB2 Call Level Interface (DB2 CLI), Open Database Connectivity (ODBC), Java Database Connectivity (JDBC), or embedded SQL for Java (SQLj) application programming interfaces.

For more information on declaring a cursor, see "DECLARE CURSOR".

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 is active for the set of *SQL-procedure-statements* that follow the set of *handler-declarations* within the compound statement in which the handler is declared, including any nested compound statements.

There are three types of condition handlers:

CONTINUE
After the handler is invoked successfully, control is returned to the SQL statement that follows the statement that raised the exception. If

the error that raised the exception is a FOR, IF, CASE, WHILE, or REPEAT statement (but not an SQL-procedure-statement within one of these), then control returns to the statement that follows END FOR, END IF, END CASE, END WHILE, or END REPEAT.

EXIT
After the handler is invoked successfully, control is returned to the end of the compound statement that declared the handler.

UNDO
Before the handler is invoked, any SQL changes that were made in the compound statement are rolled back. After the handler is invoked successfully, control is returned to the end of the compound statement that declared the handler. If UNDO is specified, the compound statement where the handler is declared must be ATOMIC.

The conditions that cause the handler to be activated are defined in the handler-declaration as follows:

specific-condition-value
Specifies that the handler is a *specific condition handler*.

SQLSTATE *string*
Specifies an SQLSTATE for which the handler is invoked. The first two characters of the SQLSTATE value must not be "00".

condition-name
Specifies a condition name for which the handler is invoked. The condition name must be previously defined in a condition declaration.

general-condition-value
Specifies that the handler is a *general condition handler*.

SQLEXCEPTION
Specifies that the handler is invoked when an exception condition occurs. An exception condition is represented by an SQLSTATE value whose 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 whose 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 whose first two characters are "02".

Rules:

- ATOMIC compound statements cannot be nested.
- The following rules apply to handler declarations:
  - A handler declaration cannot contain the same *condition-name* or SQLSTATE value more than once, and cannot contain an SQLSTATE value and a *condition-name* that represent the same SQLSTATE value.
  - Where two or more condition handlers are declared in a compound statement:
    - No two handler declarations may specify the same general condition category (SQLEXCEPTION, SQLWARNING, NOT FOUND).
    - No two handler declarations may specify the same specific condition, either as an SQLSTATE value or as a *condition-name* that represents the same value.
  - A handler is activated when it is the most appropriate handler for an exception or completion condition. The most appropriate handler is determined based on the following considerations:
    - The scope of a handler declaration *H* is the list of *SQL-procedure-statement* that follows the handler declarations contained within the compound statement in which *H* appears. This means that the scope of *H* does not include the statements contained in the body of the condition handler *H*, implying that a condition handler cannot handle conditions that arise inside its own body. Similarly, for any two handlers *H1* and *H2* declared in the same compound statement, *H1* will not handle conditions arising in the body of *H2*, and *H2* will not handle conditions arising in the body of *H1*.
    - A handler for a *specific-condition-value* or a *general-condition-value* C declared in an inner scope takes precedence over another handler for C declared in an enclosing scope.
    - When a specific handler for condition *C* and a general handler which would also handle *C* are declared in the same scope, the specific handler takes precedence over the general handler.

    If an exception condition occurs for which there is no appropriate handler, the SQL procedure containing the failing statement is terminated with an unhandled exception condition. If a completion condition occurs for which there is no appropriate handler, execution continues with the next SQL statement.

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. In the SELECT statement, casts the data type of the *salary* column from a DECIMAL into a DOUBLE.
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 `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 CAST(salary AS DOUBLE) 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
```

The following example illustrates the flow of execution in a hypothetical case where an UNDO handler is activated from another condition as the result of RESIGNAL:

```
CREATE PROCEDURE A()
LANGUAGE SQL
CS1: BEGIN ATOMIC
  DECLARE C CONDITION FOR SQLSTATE '12345';
  DECLARE D CONDITION FOR SQLSTATE '23456';

  DECLARE UNDO HANDLER FOR C
  H1: BEGIN
    -- Rollback after error, perform final cleanup, and exit
    -- procedure A.

    -- ...
```

```
    -- When this handler completes, execution continues after
    -- compound statement CS1; procedure A will terminate.
  END;

  -- Perform some work here ...
  CS2: BEGIN
    DECLARE CONTINUE HANDLER FOR D
    H2: BEGIN
      -- Perform local recovery, then forward the error
      -- condition to the outer handler for additional
      -- processing.

      -- ...

      RESIGNAL C; -- will activate UNDO handler H1; execution
                  -- WILL NOT return here.  Any local cursors
                  -- declared in H2 and CS2 will be closed.
    END;

    -- Perform some more work here ...

    -- Simulate raising of condition D by some SQL statement
    -- in compound statement CS2:
    SIGNAL D; -- will activate H2
  END;
END
```

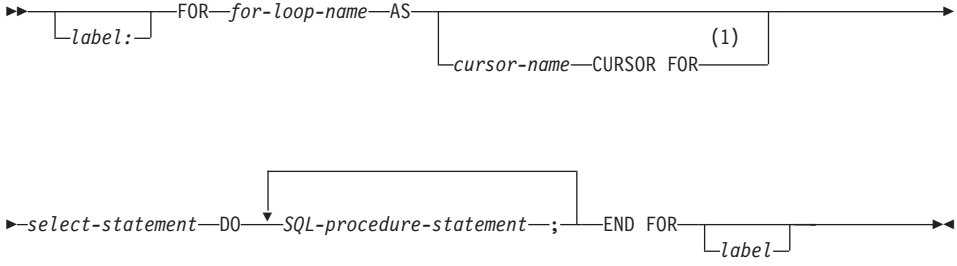
Related reference:

- "DECLARE CURSOR" on page 482
- "Data types" in the *SQL Reference, Volume 1*

FOR statement

The FOR statement executes a statement or group of statements for each row of a table.

Syntax:



Notes:

1 This option can only be used in the context of an SQL Procedure.

Description:

label
Specifies the label for the FOR statement. If the beginning label is specified, that label can be used in LEAVE and ITERATE statements. If the ending label is specified, it must be the same as the beginning label.

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 an ITERATE or LEAVE statement within the FOR statement. The *for-loop-name* is used to qualify the column names returned by the specified *select-statement*.

cursor-name
Names the cursor that is used to select rows from the result table from the SELECT statement. If not specified, DB2 generates a unique cursor name.

select-statement
Specifies the SELECT statement of the cursor. All columns in the select list must have a name and there cannot be two columns with the same name.

In a trigger, function, method, or dynamic compound statement, the *select-statement* must consist of only a *fullselect* with optional common table expressions.

SQL-procedure-statement
Specifies a statement (or statements) to be invoked for each row of the table.

Rules:

- The select list must consist of unique column names and the table specified in the select list must exist when the procedure is created, or it must be a table created in a previous SQL procedure statement.
- The cursor specified in a for-statement cannot be referenced outside the for-statement and cannot be specified in an OPEN, FETCH, or CLOSE statement.

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 ATOMIC
  DECLARE fullname CHAR(40);
  FOR v1 AS
    SELECT firstnme, midinit, lastname FROM employee
      DO
      SET fullname = lastname || ',' || firstnme ||' ' || midinit;
      INSERT INTO tnames VALUES (fullname);
  END FOR
END
```

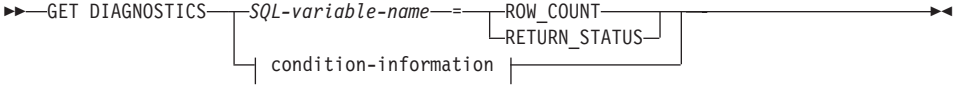
Related samples:

- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"

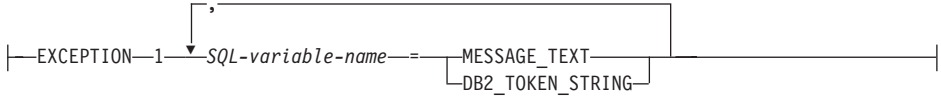
GET DIAGNOSTICS statement

The GET DIAGNOSTICS statement is used to obtain information about the previously executed SQL statement.

Syntax:



condition-information:



Description:

SQL-variable-name
Identifies the variable that is the assignment target. If ROW_COUNT or RETURN_STATUS is specified, the variable must be an integer variable. Otherwise, the variable must be CHAR or VARCHAR. SQL variables can be defined in a compound statement.

ROW_COUNT
Identifies the number of rows associated with the previous SQL statement. If the previous SQL statement is a DELETE, INSERT, or UPDATE statement, ROW_COUNT identifies the number of rows that qualified for the operation. If the previous statement is a PREPARE statement, ROW_COUNT identifies the *estimated* number of result rows in the prepared statement.

RETURN_STATUS
Identifies the status value returned from the stored procedure associated with the previously executed SQL statement, provided that the statement was a CALL statement invoking a procedure that returns a status. If the previous statement is not such a statement, then the value returned has no meaning and could be any integer.

condition-information
Specifies that the error or warning information for the previously executed SQL statement is to be returned. If information about an error is needed, the GET DIAGNOSTICS statement must be the first statement specified in the handler that will handle the error. If information about a warning is needed, and if the handler will get control of the warning condition, the GET DIAGNOSTICS statement must be the first statement specified in that handler. If the handler will *not* get control of the warning condition,

the GET DIAGNOSTICS statement must be the next statement executed. This option can only be specified in the context of an SQL Procedure (SQLSTATE 42601).

MESSAGE_TEXT
Identifies any error or warning message text returned from the previously executed SQL statement. The message text is returned in the language of the database server where the statement is processed. If the statement completed with an SQLCODE of zero, an empty string is returned for a VARCHAR variable or blanks are returned for a CHAR variable.

DB2_TOKEN_STRING
Identifies any error or warning message tokens returned from the previously executed SQL statement. If the statement completed with an SQLCODE of zero, or if the SQLCODE had no tokens, an empty string is returned for a VARCHAR variable or blanks are returned for a CHAR variable.

Notes:

- The GET DIAGNOSTICS statement does not change the contents of the diagnostics area (SQLCA). If an SQLSTATE or SQLCODE special variable is declared in the SQL procedure, these are set to the SQLSTATE or SQLCODE returned from issuing the GET DIAGNOSTICS statement.

Examples:

In an SQL procedure, execute a GET DIAGNOSTICS statement to determine how many rows were updated.

```
CREATE PROCEDURE sqlprocg (IN deptnbr VARCHAR(3))
   LANGUAGE SQL
   BEGIN
      DECLARE SQLSTATE CHAR(5);
      DECLARE rcount INTEGER;
      UPDATE CORPDATA.PROJECT
         SET PRSTAFF = PRSTAFF + 1.5
         WHERE DEPTNO = deptnbr;
      GET DIAGNOSTICS rcount = ROW_COUNT;
-- At this point, rcount contains the number of rows that were updated.
...
   END
```

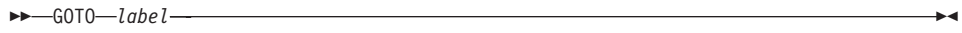
Within an SQL procedure, handle the returned status value from the invocation of a stored procedure called TRYIT that could either explicitly RETURN a positive value indicating a user failure, or encounter SQL errors that would result in a negative return status value. If the procedure is successful, it returns a value of zero.

```
CREATE PROCEDURE TESTIT ()
   LANGUAGE SQL
   A1:BEGIN
      DECLARE RETVAL INTEGER DEFAULT 0;
      ...
      CALL TRYIT;
      GET DIAGNOSTICS RETVAL = RETURN_STATUS;
      IF RETVAL <> 0 THEN
         ...
         LEAVE A1;
      ELSE
         ...
      END IF;
   END A1
```

GOTO statement

The GOTO statement is used to branch to a user-defined label within an SQL procedure.

Syntax:



Description:

label

Specifies a labelled statement where processing is to continue. The labelled statement and the GOTO statement must be in the same scope:

- If the GOTO statement is defined in a FOR statement, *label* 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, *label* 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, *label* must be defined in the same handler, following the other scope rules
- If the GOTO statement is defined outside of a handler, *label* must not be defined within a handler.

If *label* is not defined within a scope that the GOTO statement can reach, an error is returned (SQLSTATE 42736).

Notes:

- It is recommended that the GOTO statement be used sparingly. This statement interferes with normal processing sequences, 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.

Examples:

In the following compound 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.

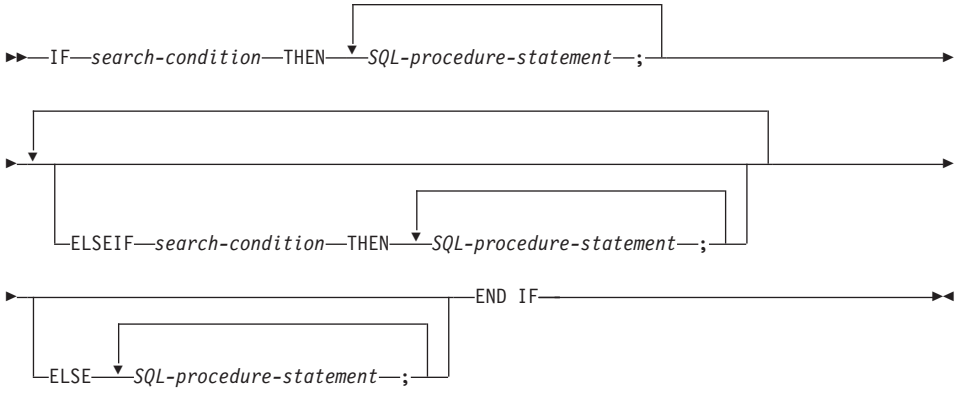
```
CREATE PROCEDURE adjust_salary
  (IN v_empno CHAR(6),
  IN rating INTEGER)
  OUT return_parm DECIMAL (8,2))
```

```
  MODIFIES SQL DATA
  LANGUAGE SQL
  BEGIN
    DECLARE new_salary DECIMAL (9,2)
    DECLARE service DECIMAL (8,2)
      SELECT SALARY, CURRENT_DATE - HIREDATE
        INTO new_salary, service
        FROM EMPLOYEE
        WHERE EMPNO = v_empno
      IF service < 600
        THEN GOTO EXIT
      END IF
      IF rating = 1
        THEN SET new_salary = new_salary + (new_salary * .10)
      ELSE IF rating = 2
        THEN SET new_salary = new_salary + (new_salary * .05)
      END IF
      UPDATE EMPLOYEE
        SET SALARY = new_salary
        WHERE EMPNO = v_empno
      EXIT: SET return_parm = service
END
```

IF statement

The IF statement selects an execution path based on the evaluation of a condition.

Syntax:



Description:

search-condition
Specifies the condition for which an SQL statement should be invoked. 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 the statement to be invoked if the preceding *search-condition* is true.

Examples:

The following SQL procedure accepts two IN parameters: an employee number *employee_number* and an employee rating *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
```

Related samples:

- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"

ITERATE statement

The ITERATE statement causes the flow of control to return to the beginning of a labelled loop.

Syntax:

Description:

label
Specifies the label of the FOR, LOOP, REPEAT, or WHILE statement to which DB2 passes the flow of control.

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
  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 loop or a compound statement.

Syntax:

Description:

label
Specifies the label of the compound, FOR, LOOP, REPEAT, or WHILE statement to exit.

Notes:

- 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
```

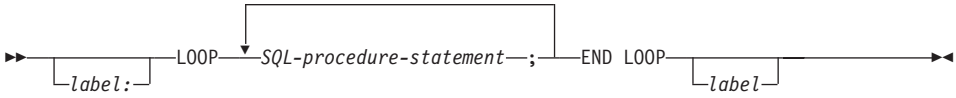
Related samples:

- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"

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 LEAVE and ITERATE statements. If the ending label is specified, a matching beginning label must be specified.

SQL-procedure-statement
Specifies the statements to be invoked in the loop.

Examples:

This procedure uses a LOOP statement to fetch values from the `employee` table. Each time the loop iterates, the OUT parameter *counter* is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes the flow of control outside of the loop.

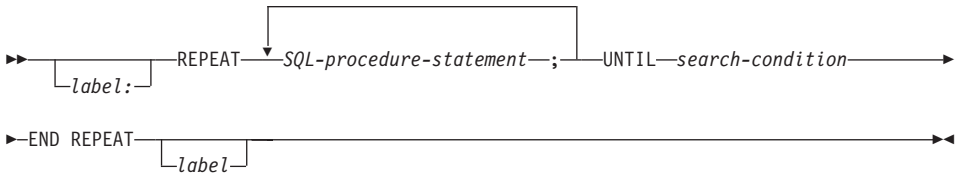
```
CREATE PROCEDURE LOOP_UNTIL_SPACE(OUT counter INTEGER)
  LANGUAGE SQL
  BEGIN
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE v_firstnme VARCHAR(12);
    DECLARE v_midinit CHAR(1);
    DECLARE v_lastname VARCHAR(15);
    DECLARE c1 CURSOR FOR
      SELECT firstnme, midinit, lastname
        FROM employee;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
      SET counter = -1;
    OPEN c1;
    fetch_loop:
    LOOP
      FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
      IF v_midinit = ' ' THEN
        LEAVE fetch_loop;
      END IF;
      SET v_counter = v_counter + 1;
```

```
  END LOOP fetch_loop;
  SET counter = v_counter;
  CLOSE c1;
END
```

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 LEAVE and ITERATE statements. If an ending label is specified, a matching beginning label also must be specified.

SQL-procedure-statement
Specifies the SQL statement to execute with the loop.

search-condition
Specifies a condition that is evaluated before each execution of the SQL procedure statement. If the condition is false, DB2 executes the SQL procedure statement in the loop.

Examples:

A REPEAT statement fetches rows from a table until the *not_found* condition handler is invoked.

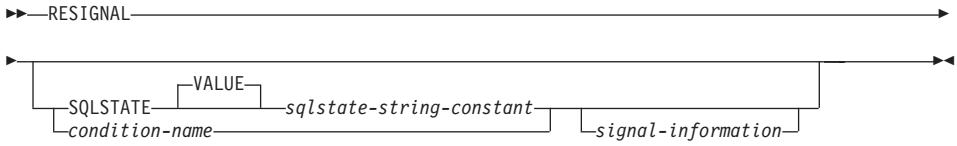
```
CREATE PROCEDURE REPEAT_STMT(OUT counter INTEGER)
  LANGUAGE SQL
  BEGIN
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE v_firstnme VARCHAR(12);
    DECLARE v_midinit CHAR(1);
    DECLARE v_lastname VARCHAR(15);
    DECLARE at_end SMALLINT DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '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
```

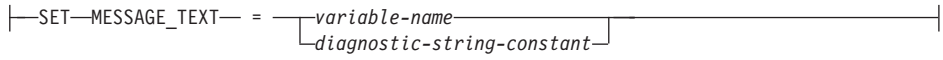
RESIGNAL statement

The RESIGNAL statement is used 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:

SQLSTATE VALUE *sqlstate-string-constant*
The specified string constant represents an SQLSTATE. It must be a character string constant with exactly 5 characters that follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z')
- The SQLSTATE class (first two characters) cannot be '00', since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is raised (SQLSTATE 428B3).

condition-name
Specifies the name of the condition.

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.

variable-name
Identifies an SQL variable that must be declared within the compound statement. The SQL variable must be defined as a CHAR or VARCHAR data type.

diagnostic-string-constant
Specifies a character string constant that contains the message text.

Notes:

- If the RESIGNAL statement is specified without an SQLSTATE clause or a *condition-name*, the identical condition that invoked the handler is returned. The SQLSTATE, SQLCODE and the SQLCA associated with the condition are unchanged.
- If a RESIGNAL statement is issued, and an SQLSTATE or *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 condition is returned and the SQLCODE is set to +438.
  - Otherwise, an exception condition is returned and the SQLCODE is set to -438.

  The other fields of the SQLCA are set as follows:
  - sqlerrd fields are set to zero
  - sqlwarn fields are set to blank
  - sqlerrmc is set to the first 70 bytes of MESSAGE_TEXT
  - sqlerrml is set to the length of sqlerrmc, or to zero if no SET MESSAGE_TEXT clause is specified
  - sqlerrp is set to ROUTINE.
- Refer to the ″Notes″ section under ″SIGNAL statement″ for further information on SQLSTATE values.

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 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 result = numerator / denominator;
    END IF;
END
```

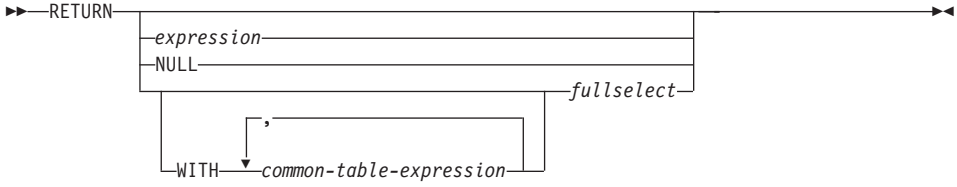
Related reference:

- "SIGNAL statement" on page 796

RETURN statement

The RETURN statement is used to return from the routine. For SQL functions or methods, it returns the result of the function or method. For an SQL procedure, it optionally returns an integer status value.

Syntax:



Description:

expression
Specifies a value that is returned from the routine:

- If the routine is a function or method, one of *expression*, NULL, or *fullselect* must be specified (SQLSTATE 42630) and the data type of the result must be assignable to the RETURNS type of the routine (SQLSTATE 42866).
- If the routine is a table function, a scalar expression (other than a scalar fullselect) cannot be specified (SQLSTATE 428F1).
- If the routine is a procedure, the data type of *expression* must be INTEGER (SQLSTATE 428E2). A procedure cannot return NULL or a *fullselect*.

NULL
Specifies that the function or method returns a null value of the data type defined in the RETURNS clause. NULL cannot be specified for a RETURN from a procedure.

WITH *common-table-expression*
Defines a common table expression for use with the *fullselect* that follows.

fullselect
Specifies the row or rows to be returned for the function. The number of columns in the *fullselect* must match the number of columns in the function result (SQLSTATE 42811). In addition, the static column types of the *fullselect* must be assignable to the declared column types of the function result, using the rules for assignment to columns (SQLSTATE 42866).

The *fullselect* cannot be specified for a RETURN from a procedure.

If the routine is a scalar function or method, then the *fullselect* must return one column (SQLSTATE 42823) and, at most, one row (SQLSTATE 21000).

If the routine is a row function, it must return, at most, one row (SQLSTATE 21505). However, one or more columns can be returned.

If the routine is a table function, it can return zero or more rows with one or more columns.

Rules:

- The execution of an SQL function or method must end with a RETURN statement (SQLSTATE 42632).
- In an SQL table or row function using a *dynamic-compound-statement*, the only RETURN statement allowed is the one at the end of the compound statement. (SQLSTATE 429BD).

Notes:

- When a value is returned from a procedure, the caller may access the value:
  - using the GET DIAGNOSTICS statement to retrieve the RETURN_STATUS when the SQL procedure was called from another SQL procedure
  - using the parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a CLI application
  - directly from the sqlerrd[0] field of the SQLCA, after processing the CALL of an SQL procedure. This field is only valid if the SQLCODE is zero or positive (assume a value of –1 otherwise).

Examples:

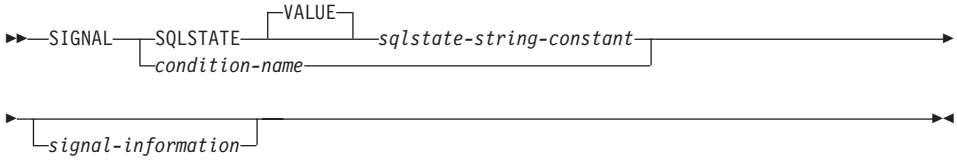
Use a RETURN statement to return from an SQL stored procedure with a status value of zero if successful, and –200 if not.

```
BEGIN
...
  GOTO FAIL
...
  SUCCESS: RETURN 0
  FAIL: RETURN -200
END
```

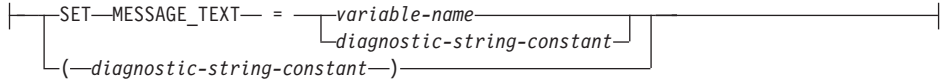
SIGNAL statement

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:

SQLSTATE VALUE *sqlstate-string-constant*
The specified string constant represents an SQLSTATE. It must be a character string constant with exactly 5 characters that follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00', since this represents successful completion.

In the context of either a dynamic compound statement, trigger, SQL function, or SQL method, the following rules must also be applied:

- The SQLSTATE class (first two characters) cannot be '01' or '02', since these are not error classes.
- If the SQLSTATE class starts with the numbers '0' through '6' or the letters 'A' through 'H', then the subclass (the last three characters) must start with a letter in the range of 'I' through 'Z'.
- If the SQLSTATE class starts with the numbers '7', '8', '9', or the letters 'I' through 'Z', then the subclass can be any of '0' through '9' or 'A' through 'Z'.

If the SQLSTATE does not conform to these rules, an error is raised (SQLSTATE 428B3).

condition-name
Specifies the name of the condition. The condition name must be unique within the procedure and can only be referenced within the compound statement in which it is declared.

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.

variable-name
Identifies an SQL variable that must be declared within the compound statement. The SQL variable must be defined as a CHAR or VARCHAR data type.

diagnostic-string-constant
Specifies a character string constant that contains the message text.

diagnostic-string
An expression with a type of CHAR or VARCHAR that returns a character string of up to 70 bytes to describe the error condition. If the string is longer than 70 bytes, it will be truncated. This option is only provided within the scope of a CREATE TRIGGER statement, for compatibility with older versions of DB2. Regular use is not recommended.

Notes:

- If a SIGNAL statement is issued, the SQLCODE returned is based on the SQLSTATE as follows:
  - If the specified SQLSTATE class is either '01' or '02', a warning or not found condition is returned and the SQLCODE is set to +438.
  - Otherwise, an exception condition is returned and the SQLCODE is set to -438.

  The other fields of the SQLCA are set as follows:
  - sqlerrd fields are set to zero
  - sqlwarn fields are set to blank
  - sqlerrmc is set to the first 70 bytes of MESSAGE_TEXT
  - sqlerrml is set to the length of sqlerrmc, or to zero if no SET MESSAGE_TEXT clause is specified
  - sqlerrp is set to ROUTINE.
- 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.

  Any valid SQLSTATE value can be used in the SIGNAL statement. However, it is recommended that programmers define new SQLSTATEs

based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

- SQLSTATE 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.

Examples:

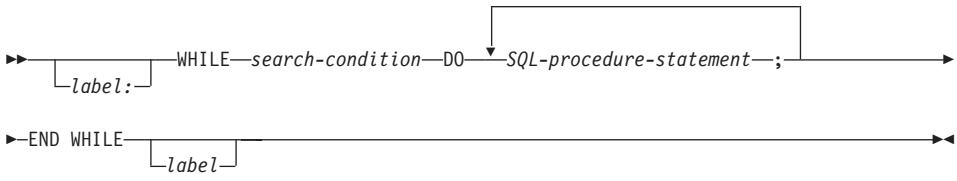
An SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
  (IN ONUM INTEGER, IN CNUM INTEGER,
   IN PNUM INTEGER, IN QNUM INTEGER)
   SPECIFIC SUBMIT_ORDER
   MODIFIES SQL DATA
  LANGUAGE SQL
  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 LEAVE and ITERATE statements. If the ending label is specified, it must be the same as the beginning label.

search-condition
Specifies a condition that is evaluated before each execution of the loop. If the condition is true, the SQL-procedure-statements in the loop are processed.

SQL-procedure-statement
Specifies the SQL statement or statements to execute within the loop.

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 DOUBLE)
  LANGUAGE SQL
  BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE c1 CURSOR FOR
      SELECT CAST(salary AS DOUBLE)
        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
```

Related samples:

- "dbinline.sqc -- How to use inline SQL Procedure Language (C)"

Appendix A. DB2 Universal Database technical information

Overview of DB2 Universal Database technical information

DB2 Universal Database technical information can be obtained in the following formats:

- Books (PDF and hard-copy formats)
- A topic tree (HTML format)
- Help for DB2 tools (HTML format)
- Sample programs (HTML format)
- Command line help
- Tutorials

This section is an overview of the technical information that is provided and how you can access it.

Categories of DB2 technical information

The DB2 technical information is categorized by the following headings:

- Core DB2 information
- Administration information
- Application development information
- Business intelligence information
- DB2 Connect information
- Getting started information
- Tutorial information
- Optional component information
- Release notes

The following tables describe, for each book in the DB2 library, the information needed to order the hard copy, print or view the PDF, or locate the HTML directory for that book. A full description of each of the books in the DB2 library is available from the IBM Publications Center at www.ibm.com/shop/publications/order

The installation directory for the HTML documentation CD differs for each category of information:

htmlcdpath/doc/htmlcd/%L/*category*

where:

- *htmlcdpath* is the directory where the HTML CD is installed.
- *%L* is the language identifier. For example, en_US.
- *category* is the category identifier. For example, core for the core DB2 information.

In the PDF file name column in the following tables, the character in the sixth position of the file name indicates the language version of a book. For example, the file name `db2d1e80` identifies the English version of the *Administration Guide: Planning* and the file name `db2d1g80` identifies the German version of the same book. The following letters are used in the sixth position of the file name to indicate the language version:

Language	**Identifier**
Arabic	w
Brazilian Portuguese	b
Bulgarian	u
Croatian	9
Czech	x
Danish	d
Dutch	q
English	e
Finnish	y
French	f
German	g
Greek	a
Hungarian	h
Italian	i
Japanese	j
Korean	k
Norwegian	n
Polish	p
Portuguese	v
Romanian	8
Russian	r
Simp. Chinese	c
Slovakian	7
Slovenian	l
Spanish	z
Swedish	s
Trad. Chinese	t
Turkish	m

No form number indicates that the book is only available online and does not have a printed version.

Core DB2 information

The information in this category cover DB2 topics that are fundamental to all DB2 users. You will find the information in this category useful whether you are a programmer, a database administrator, or you work with DB2 Connect, DB2 Warehouse Manager, or other DB2 products.

The installation directory for this category is doc/htmlcd/%L/core.

Table 14. Core DB2 information

Name	Form Number	PDF File Name
IBM DB2 Universal Database Command Reference	SC09-4828	db2n0x80
IBM DB2 Universal Database Glossary	No form number	db2t0x80
IBM DB2 Universal Database Master Index	SC09-4839	db2w0x80
IBM DB2 Universal Database Message Reference, Volume 1	GC09-4840	db2m1x80
IBM DB2 Universal Database Message Reference, Volume 2	GC09-4841	db2m2x80
IBM DB2 Universal Database What's New	SC09-4848	db2q0x80

Administration information

The information in this category covers those topics required to effectively design, implement, and maintain DB2 databases, data warehouses, and federated systems.

The installation directory for this category is doc/htmlcd/%L/admin.

Table 15. Administration information

Name	Form number	PDF file name
IBM DB2 Universal Database Administration Guide: Planning	SC09-4822	db2d1x80
IBM DB2 Universal Database Administration Guide: Implementation	SC09-4820	db2d2x80
IBM DB2 Universal Database Administration Guide: Performance	SC09-4821	db2d3x80
IBM DB2 Universal Database Administrative API Reference	SC09-4824	db2b0x80

Table 15. Administration information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Data Movement Utilities Guide and Reference	SC09-4830	db2dmx80
IBM DB2 Universal Database Data Recovery and High Availability Guide and Reference	SC09-4831	db2hax80
IBM DB2 Universal Database Data Warehouse Center Administration Guide	SC27-1123	db2ddx80
IBM DB2 Universal Database Federated Systems Guide	GC27-1224	db2fpx80
IBM DB2 Universal Database Guide to GUI Tools for Administration and Development	SC09-4851	db2atx80
IBM DB2 Universal Database Replication Guide and Reference	SC27-1121	db2e0x80
IBM DB2 Installing and Administering a Satellite Environment	GC09-4823	db2dsx80
IBM DB2 Universal Database SQL Reference, Volume 1	SC09-4844	db2s1x80
IBM DB2 Universal Database SQL Reference, Volume 2	SC09-4845	db2s2x80
IBM DB2 Universal Database System Monitor Guide and Reference	SC09-4847	db2f0x80

Application development information

The information in this category is of special interest to application developers or programmers working with DB2. You will find information about supported languages and compilers, as well as the documentation required to access DB2 using the various supported programming interfaces, such as embedded SQL, ODBC, JDBC, SQLj, and CLI. If you view this information online in HTML you can also access a set of DB2 sample programs in HTML.

The installation directory for this category is doc/htmlcd/%L/ad.

Table 16. Application development information

Name	Form number	PDF file name
IBM DB2 Universal Database Application Development Guide: Building and Running Applications	SC09-4825	db2axx80
IBM DB2 Universal Database Application Development Guide: Programming Client Applications	SC09-4826	db2a1x80
IBM DB2 Universal Database Application Development Guide: Programming Server Applications	SC09-4827	db2a2x80
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 1	SC09-4849	db2l1x80
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 2	SC09-4850	db2l2x80
IBM DB2 Universal Database Data Warehouse Center Application Integration Guide	SC27-1124	db2adx80
IBM DB2 XML Extender Administration and Programming	SC27-1234	db2sxx80

Business intelligence information

The information in this category describes how to use components that enhance the data warehousing and analytical capabilities of DB2 Universal Database.

The installation directory for this category is doc/htmlcd/%L/wareh.

Table 17. Business intelligence information

Name	Form number	PDF file name
IBM DB2 Warehouse Manager Information Catalog Center Administration Guide	SC27-1125	db2dix80
IBM DB2 Warehouse Manager Installation Guide	GC27-1122	db2idx80

DB2 Connect information

The information in this category describes how to access host or iSeries data using DB2 Connect Enterprise Edition or DB2 Connect Personal Edition.

The installation directory for this category is doc/htmlcd/%L/conn.

Table 18. DB2 Connect information

Name	Form number	PDF file name
APPC, CPI-C, and SNA Sense Codes	No form number	db2apx80
IBM Connectivity Supplement	No form number	db2h1x80
IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition	GC09-4833	db2c6x80
IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition	GC09-4834	db2c1x80
IBM DB2 Connect User's Guide	SC09-4835	db2c0x80

Getting started information

The information in this category is useful when you are installing and configuring servers, clients, and other DB2 products.

The installation directory for this category is doc/htmlcd/%L/start.

Table 19. Getting started information

Name	Form number	PDF file name
IBM DB2 Universal Database Quick Beginnings for DB2 Clients	GC09-4832	db2itx80
IBM DB2 Universal Database Quick Beginnings for DB2 Servers	GC09-4836	db2isx80
IBM DB2 Universal Database Quick Beginnings for DB2 Personal Edition	GC09-4838	db2i1x80
IBM DB2 Universal Database Installation and Configuration Supplement	GC09-4837	db2iyx80
IBM DB2 Universal Database Quick Beginnings for DB2 Data Links Manager	GC09-4829	db2z6x80

Tutorial information

Tutorial information introduces DB2 features and teaches how to perform various tasks.

The installation directory for this category is doc/htmlcd/%L/tutr.

Table 20. Tutorial information

Name	Form number	PDF file name
Business Intelligence Tutorial: Introduction to the Data Warehouse	No form number	db2tux80
Business Intelligence Tutorial: Extended Lessons in Data Warehousing	No form number	db2tax80
Development Center Tutorial for Video Online using Microsoft Visual Basic	No form number	db2tdx80
Information Catalog Center Tutorial	No form number	db2aix80
Video Central for e-business Tutorial	No form number	db2twx80
Visual Explain Tutorial	No form number	db2tvx80

Optional component information

The information in this category describes how to work with optional DB2 components.

The installation directory for this category is doc/htmlcd/%L/opt.

Table 21. Optional component information

Name	Form number	PDF file name
IBM DB2 Life Sciences Data Connect Planning, Installation, and Configuration Guide	GC27-1235	db2lsx80
IBM DB2 Spatial Extender User's Guide and Reference	SC27-1226	db2sbx80
IBM DB2 Universal Database Data Links Manager Administration Guide and Reference	SC27-1221	db2z0x80

Table 21. Optional component information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Net Search Extender Administration and Programming Guide **Note:** HTML for this document is not installed from the HTML documentation CD.	SH12-6740	N/A

Release notes

The release notes provide additional information specific to your product's release and FixPak level. They also provides summaries of the documentation updates incorporated in each release and FixPak.

Table 22. Release notes

Name	Form number	PDF file name	HTML directory
DB2 Release Notes	See note.	See note.	doc/prodcd/%L/db2ir where %L is the language identifier.
DB2 Connect Release Notes	See note.	See note.	doc/prodcd/%L/db2cr where %L is the language identifier.
DB2 Installation Notes	Available on product CD-ROM only.	Available on product CD-ROM only.	

Note: The HTML version of the release notes is available from the Information Center and on the product CD-ROMs. To view the ASCII file:

- On UNIX-based platforms, see the Release.Notes file. This file is located in the DB2DIR/Readme/%L directory, where %L represents the locale name and DB2DIR represents:
  - /usr/opt/db2_08_01 on AIX
  - /opt/IBM/db2/V8.1 on all other UNIX operating systems
- On other platforms, see the RELEASE.TXT file. This file is located in the directory where the product is installed.

Related tasks:

- "Printing DB2 books from PDF files" on page 809

- "Ordering printed DB2 books" on page 810
- "Accessing online help" on page 810
- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 814
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 815

Printing DB2 books from PDF files

You can print DB2 books from the PDF files on the *DB2 PDF Documentation* CD. Using Adobe Acrobat Reader, you can print either the entire book or a specific range of pages.

Prerequisites:

Ensure that you have Adobe Acrobat Reader. It is available from the Adobe Web site at www.adobe.com

Procedure:

To print a DB2 book from a PDF file:

1. Insert the *DB2 PDF Documentation* CD. On UNIX operating systems, mount the DB2 PDF Documentation CD. Refer to your *Quick Beginnings* book for details on how to mount a CD on UNIX operating systems.
2. Start Adobe Acrobat Reader.
3. Open the PDF file from one of the following locations:
   - On Windows operating systems:

     x:\doc*language* directory, where *x* represents the CD-ROM drive letter and *language* represents the two-character territory code that represents your language (for example, EN for English).
   - On UNIX operating systems:

     /cdrom/doc/*%L* directory on the CD-ROM, where */cdrom* represents the mount point of the CD-ROM and *%L* represents the name of the desired locale.

Related tasks:

- "Ordering printed DB2 books" on page 810
- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 814
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 815

Related reference:

- "Overview of DB2 Universal Database technical information" on page 801

Ordering printed DB2 books

Procedure:

To order printed books:

- Contact your IBM authorized dealer or marketing representative. To find a local IBM representative, check the IBM Worldwide Directory of Contacts at www.ibm.com/shop/planetwide
- Phone `1-800-879-2755` in the United States or `1-800-IBM-4YOU` in Canada.
- Visit the IBM Publications Center at www.ibm.com/shop/publications/order

Related tasks:

- "Printing DB2 books from PDF files" on page 809
- "Finding topics by accessing the DB2 Information Center from a browser" on page 812
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 815

Related reference:

- "Overview of DB2 Universal Database technical information" on page 801

Accessing online help

The online help that comes with all DB2 components is available in three types:

- Window and notebook help
- Command line help
- SQL statement help

Window and notebook help explain the tasks that you can perform in a window or notebook and describe the controls. This help has two types:

- Help accessible from the **Help** button
- Infopops

The **Help** button gives you access to overview and prerequisite information. The infopops describe the controls in the window or notebook. Window and notebook help are available from DB2 centers and components that have user interfaces.

Command line help includes Command help and Message help. Command help explains the syntax of commands in the command line processor. Message help describes the cause of an error message and describes any action you should take in response to the error.

SQL statement help includes SQL help and SQLSTATE help. DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the syntax of SQL statements (SQL states and class codes).

Note: SQL help is not available for UNIX operating systems.

Procedure:

To access online help:

- For window and notebook help, click **Help** or click that control, then click **F1**. If the **Automatically display infopops** check box on the **General** page of the **Tool Settings** notebook is selected, you can also see the infopop for a particular control by holding the mouse cursor over the control.
- For command line help, open the command line processor and enter:
  - For Command help:

    ```
    ? command
    ```

    where *command* represents a keyword or the entire command.

    For example, `? catalog` displays help for all the CATALOG commands, while `? catalog database` displays help for the CATALOG DATABASE command.
- For Message help:

  ```
  ? XXXnnnnn
  ```

  where *XXXnnnnn* represents a valid message identifier.

  For example, `? SQL30081` displays help about the SQL30081 message.
- For SQL statement help, open the command line processor and enter:
  - For SQL help:

    ```
    ? sqlstate or ? class code
    ```

    where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

    For example, `? 08003` displays help for the 08003 SQL state, while `? 08` displays help for the 08 class code.
  - For SQLSTATE help:

```
help statement
```

where *statement* represents an SQL statement.

For example, help SELECT displays help about the SELECT statement.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 812
- "Viewing technical documentation online directly from the DB2 HTML Documentation CD" on page 815

Finding topics by accessing the DB2 Information Center from a browser

The DB2 Information Center accessed from a browser enables you to access the information you need to take full advantage of DB2 Universal Database and DB2 Connect. The DB2 Information Center also documents major DB2 features and components including replication, data warehousing, metadata, Life Sciences Data Connect, and DB2 extenders.

The DB2 Information Center accessed from a browser is composed of the following major elements:

Navigation tree
The navigation tree is located in the left frame of the browser window. The tree expands and collapses to show and hide topics, the glossary, and the master index in the DB2 Information Center.

Navigation toolbar
The navigation toolbar is located in the top right frame of the browser window. The navigation toolbar contains buttons that enable you to search the DB2 Information Center, hide the navigation tree, and find the currently displayed topic in the navigation tree.

Content frame
The content frame is located in the bottom right frame of the browser window. The content frame displays topics from the DB2 Information Center when you click on a link in the navigation tree, click on a search result, or follow a link from another topic or from the master index.

Prerequisites:

To access the DB2 Information Center from a browser, you must use one of the following browsers:

- Microsoft Explorer, version 5 or later
- Netscape Navigator, version 6.1 or later

Restrictions:

The DB2 Information Center contains only those sets of topics that you chose to install from the *DB2 HTML Documentation CD*. If your Web browser returns a `File not found` error when you try to follow a link to a topic, you must install one or more additional sets of topics *DB2 HTML Documentation CD*.

Procedure:

To find a topic by searching with keywords:

1. In the navigation toolbar, click **Search**.
2. In the top text entry field of the Search window, enter two or more terms related to your area of interest and click **Search**. A list of topics ranked by accuracy displays in the **Results** field.

   Entering more terms increases the precision of your query while reducing the number of topics returned from your query.
3. In the **Results** field, click the title of the topic you want to read. The topic displays in the content frame.

To find a topic in the navigation tree:

1. In the navigation tree, click the book icon of the category of topics related to your area of interest. A list of subcategories displays underneath the icon.
2. Continue to click the book icons until you find the category containing the topics in which you are interested. Categories that link to topics display the category title as an underscored link when you move the cursor over the category title. The navigation tree identifies topics with a page icon.
3. Click the topic link. The topic displays in the content frame.

To find a topic or term in the master index:

1. In the navigation tree, click the “Index” category. The category expands to display a list of links arranged in alphabetical order in the navigation tree.
2. In the navigation tree, click the link corresponding to the first character of the term relating to the topic in which you are interested. A list of terms with that initial character displays in the content frame. Terms that have multiple index entries are identified by a book icon.
3. Click the book icon corresponding to the term in which you are interested. A list of subterms and topics displays below the term you clicked. Topics are identified by page icons with an underscored title.
4. Click on the title of the topic that meets your needs. The topic displays in the content frame.

Related concepts:

- "Accessibility" on page 821
- "DB2 Information Center for topics" on page 823

Related tasks:

- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 814
- "Updating the HTML documentation installed on your machine" on page 816
- "Troubleshooting DB2 documentation search with Netscape 4.x" on page 818
- "Searching the DB2 documentation" on page 819

Related reference:

- "Overview of DB2 Universal Database technical information" on page 801

Finding product information by accessing the DB2 Information Center from the administration tools

The DB2 Information Center provides quick access to DB2 product information and is available on all operating systems for which the DB2 administration tools are available.

The DB2 Information Center accessed from the tools provides six types of information.

Tasks Key tasks you can perform using DB2.

Concepts
Key concepts for DB2.

Reference
DB2 reference information, such as keywords, commands, and APIs.

Troubleshooting
Error messages and information to help you with common DB2 problems.

Samples
Links to HTML listings of the sample programs provided with DB2.

Tutorials
Instructional aid designed to help you learn a DB2 feature.

Prerequisites:

Some links in the DB2 Information Center point to Web sites on the Internet. To display the content for these links, you will first have to connect to the Internet.

Procedure:

To find product information by accessing the DB2 Information Center from the tools:

1. Start the DB2 Information Center in one of the following ways:
   - From the graphical administration tools, click on the **Information Center** icon in the toolbar. You can also select it from the **Help** menu.
   - At the command line, enter **db2ic**.
2. Click the tab of the information type related to the information you are attempting to find.
3. Navigate through the tree and click on the topic in which you are interested. The Information Center will then launch a Web browser to display the information.
4. To find information without browsing the lists, click the **Search** icon to the right of the list.

   Once the Information Center has launched a browser to display the information, you can perform a full-text search by clicking the **Search** icon in the navigation toolbar.

Related concepts:

- "Accessibility" on page 821
- "DB2 Information Center for topics" on page 823

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 812
- "Searching the DB2 documentation" on page 819

Viewing technical documentation online directly from the DB2 HTML Documentation CD

All of the HTML topics that you can install from the *DB2 HTML Documentation CD* can also be read directly from the CD. Therefore, you can view the documentation without having to install it.

Restrictions:

Because the following items are installed from the DB2 product CD and not the *DB2 HTML Documentation CD*, you must install the DB2 product to view these items:

- Tools help
- DB2 Quick Tour
- Release notes

Procedure:

1. Insert the *DB2 HTML Documentation* CD. On UNIX operating systems, mount the *DB2 HTML Documentation CD*. Refer to your *Quick Beginnings* book for details on how to mount a CD on UNIX operating systems.
2. Start your HTML browser and open the appropriate file:
   - For Windows operating systems:

     ```
     e:\Program Files\sqllib\doc\htmlcd\%L\index.htm
     ```

     where *e* represents the CD-ROM drive, and %L is the locale of the documentation that you wish to use, for example, **en_US** for English.
   - For UNIX operating systems:

     ```
     /cdrom/Program Files/sqllib/doc/htmlcd/%L/index.htm
     ```

     where */cdrom/* represents where the CD is mounted, and %L is the locale of the documentation that you wish to use, for example, **en_US** for English.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 812
- "Copying files from the DB2 HTML Documentation CD to a Web Server" on page 818

Related reference:

- "Overview of DB2 Universal Database technical information" on page 801

Updating the HTML documentation installed on your machine

It is now possible to update the HTML installed from the *DB2 HTML Documentation CD* when updates are made available from IBM. This can be done in one of two ways:

- Using the Information Center (if you have the DB2 administration GUI tools installed).
- By downloading and applying a DB2 HTML documentation FixPak .

Note: This will NOT update the DB2 code; it will only update the HTML documentation installed from the *DB2 HTML Documentation CD*.

Procedure:

To use the Information Center to update your local documentation:

1. Start the DB2 Information Center in one of the following ways:
   - From the graphical administration tools, click on the **Information Center** icon in the toolbar. You can also select it from the **Help** menu.
   - At the command line, enter **db2ic**.
2. Ensure your machine has access to the external Internet; the updater will download the latest documentation FixPak from the IBM server if required.
3. Select **Information Center —> Update Local Documentation** from the menu to start the update.
4. Supply your proxy information (if required) to connect to the external Internet.

The Information Center will download and apply the latest documentation FixPak, if one is available.

To manually download and apply the documentation FixPak :

1. Ensure your machine is connected to the Internet.
2. Open the DB2 support page in your Web browser at: www.ibm.com/software/data/db2/udb/winos2unix/support
3. Follow the link for version 8 and look for the ″Documentation FixPaks″ link.
4. Determine if the version of your local documentation is out of date by comparing the documentation FixPak level to the documentation level you have installed. This current documentation on your machine is at the following level: **DB2 v8.1 GA**.
5. If there is a more recent version of the documentation available then download the FixPak applicable to your operating system. There is one FixPak for all Windows platforms, and one FixPak for all UNIX platforms.
6. Apply the FixPak:
   - For Windows operating systems: The documentation FixPak is a self extracting zip file. Place the downloaded documentation FixPak in an empty directory, and run it. It will create a **setup** command which you can run to install the documentation FixPak.
   - For UNIX operating systems: The documentation FixPak is a compressed tar.Z file. Uncompress and untar the file. It will create a directory named `delta_install` with a script called **installdocfix**. Run this script to install the documentation FixPak.

Related tasks:

- "Copying files from the DB2 HTML Documentation CD to a Web Server" on page 818

Related reference:

- "Overview of DB2 Universal Database technical information" on page 801

Copying files from the DB2 HTML Documentation CD to a Web Server

The entire DB2 information library is delivered to you on the *DB2 HTML Documentation CD*, so you can install the library on a Web server for easier access. Simply copy to your Web server the documentation for the languages that you want.

Procedure:

To copy files from the *DB2 HTML Documentation CD* to a Web server, use the appropriate path:

- For Windows operating systems:

  ```
  E:\Program Files\sqllib\doc\htmlcd\%L\*.*
  ```

  where *E* represents the CD-ROM drive and *%L* represents the language identifier.
- For UNIX operating systems:

  ```
  /cdrom:Program Files/sqllib/doc/htmlcd/%L/*.*
  ```

  where *cdrom* represents the CD-ROM drive and *%L* represents the language identifier.

Related tasks:

- "Searching the DB2 documentation" on page 819

Related reference:

- "Supported DB2 interface languages, locales, and code pages" in the *Quick Beginnings for DB2 Servers*
- "Overview of DB2 Universal Database technical information" on page 801

Troubleshooting DB2 documentation search with Netscape 4.x

Most search problems are related to the Java support provided by web browsers. This task describes possible workarounds.

Procedure:

A common problem with Netscape 4.x involves a missing or misplaced security class. Try the following workaround, especially if you see the following line in the browser Java console:

```
Cannot find class  java/security/InvalidParameterException
```

- On Windows operating systems:

  From the *DB2 HTML Documentation CD*, copy the supplied *x*:Program Files\sqllib\doc\htmlcd*locale*\InvalidParameterException.class file to the java\classes\java\security\ directory relative to your Netscape browser installation, where *x* represents the CD-ROM drive letter and *locale* represents the name of the desired locale.

  Note: You may have to create the java\security\ subdirectory structure.

- On UNIX operating systems:

  From the *DB2 HTML Documentation CD*, copy the supplied */cdrom*/Program Files/sqllib/doc/htmlcd/*locale*/InvalidParameterException.class file to the java/classes/java/security/ directory relative to your Netscape browser installation, where *cdrom* represents the mount point of the CD-ROM and *locale* represents the name of the desired locale.

  Note: You may have to create the java/security/ subdirectory structure.

If your Netscape browser still fails to display the search input window, try the following:

- Stop all instances of Netscape browsers to ensure that there is no Netscape code running on the machine. Then open a new instance of the Netscape browser and try to start the search again.
- Purge the browser's cache.
- Try a different version of Netscape, or a different browser.

Related tasks:

- "Searching the DB2 documentation" on page 819

Searching the DB2 documentation

To search DB2's documentation, you need Netscape 6.1 or higher, or Microsoft's Internet Explorer 5 or higher. Ensure that your browser's Java support is enabled.

A pop-up search window opens when you click the search icon in the navigation toolbar of the Information Center accessed from a browser. If you are using the search for the first time it may take a minute or so to load into the search window.

Restrictions:

The following restrictions apply when you use the documentation search:

- Boolean searches are not supported. The boolean search qualifiers *and* and *or* will be ignored in a search. For example, the following searches would produce the same results:
  - servlets *and* beans
  - servlets *or* beans
- Wildcard searches are not supported. A search on *java** will only look for the literal string *java** and would not, for example, find *javadoc*.

In general, you will get better search results if you search for phrases instead of single words.

Procedure:

To search the DB2 documentation:

1. In the navigation toolbar, click **Search**.
2. In the top text entry field of the Search window, enter two or more terms related to your area of interest and click **Search**. A list of topics ranked by accuracy displays in the **Results** field.

   Entering more terms increases the precision of your query while reducing the number of topics returned from your query.
3. In the **Results** field, click the title of the topic you want to read. The topic displays in the content frame.

Note: When you perform a search, the first result is automatically loaded into your browser frame. To view the contents of other search results, click on the result in results lists.

Related tasks:

- "Troubleshooting DB2 documentation search with Netscape 4.x" on page 818

Online DB2 troubleshooting information

With the release of DB2® UDB Version 8, there will no longer be a *Troubleshooting Guide*. The troubleshooting information once contained in this guide has been integrated into the DB2 publications. By doing this, we are able to deliver the most up-to-date information possible. To find information on the troubleshooting utilities and functions of DB2, access the DB2 Information Center from any of the tools.

Refer to the DB2 Online Support site if you are experiencing problems and want help finding possible causes and solutions. The support site contains a

large, constantly updated database of DB2 publications, TechNotes, APAR (product problem) records, FixPaks, and other resources. You can use the support site to search through this knowledge base and find possible solutions to your problems.

Access the Online Support site at www.ibm.com/software/data/db2/udb/winos2unix/support, or by clicking the **Online Support** button in the DB2 Information Center. Frequently changing information, such as the listing of internal DB2 error codes, is now also available from this site.

Related concepts:

- "DB2 Information Center for topics" on page 823

Related tasks:

- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 814

Accessibility

Accessibility features help users with physical disabilities, such as restricted mobility or limited vision, to use software products successfully. These are the major accessibility features in DB2® Universal Database Version 8:

- DB2 allows you to operate all features using the keyboard instead of the mouse. See "Keyboard Input and Navigation".
- DB2 enables you customize the size and color of your fonts. See "Accessible Display" on page 822.
- DB2 allows you to receive either visual or audio alert cues. See "Alternative Alert Cues" on page 822.
- DB2 supports accessibility applications that use the Java™ Accessibility API. See "Compatibility with Assistive Technologies" on page 822.
- DB2 comes with documentation that is provided in an accessible format. See "Accessible Documentation" on page 822.

Keyboard Input and Navigation

Keyboard Input

You can operate the DB2 Tools using only the keyboard. You can use keys or key combinations to perform most operations that can also be done using a mouse.

Keyboard Focus

In UNIX-based systems, the position of the keyboard focus is highlighted, indicating which area of the window is active and where your keystrokes will have an effect.

Accessible Display

The DB2 Tools have features that enhance the user interface and improve accessibility for users with low vision. These accessibility enhancements include support for customizable font properties.

Font Settings

The DB2 Tools allow you to select the color, size, and font for the text in menus and dialog windows, using the Tools Settings notebook.

Non-dependence on Color

You do not need to distinguish between colors in order to use any of the functions in this product.

Alternative Alert Cues

You can specify whether you want to receive alerts through audio or visual cues, using the Tools Settings notebook.

Compatibility with Assistive Technologies

The DB2 Tools interface supports the Java Accessibility API enabling use by screen readers and other assistive technologies used by people with disabilities.

Accessible Documentation

Documentation for the DB2 family of products is available in HTML format. This allows you to view documentation according to the display preferences set in your browser. It also allows you to use screen readers and other assistive technologies.

DB2 tutorials

The DB2® tutorials help you learn about various aspects of DB2 Universal Database. The tutorials provide lessons with step-by-step instructions in the areas of developing applications, tuning SQL query performance, working with data warehouses, managing metadata, and developing Web services using DB2.

Before you begin:

Before you can access these tutorials using the links below, you must install the tutorials from the *DB2 HTML Documentation* CD-ROM.

If you do not want to install the tutorials, you can view the HTML versions of the tutorials directly from the *DB2 HTML Documentation CD*. PDF versions of these tutorials are also available on the *DB2 PDF Documentation CD*.

Some tutorial lessons use sample data or code. See each individual tutorial for a description of any prerequisites for its specific tasks.

DB2 Universal Database tutorials:

If you installed the tutorials from the *DB2 HTML Documentation* CD-ROM, you can click on a tutorial title in the following list to view that tutorial.

Business Intelligence Tutorial: Introduction to the Data Warehouse Center
Perform introductory data warehousing tasks using the Data Warehouse Center.

Business Intelligence Tutorial: Extended Lessons in Data Warehousing
Perform advanced data warehousing tasks using the Data Warehouse Center.

Development Center Tutorial for Video Online using Microsoft® Visual Basic
Build various components of an application using the Development Center Add-in for Microsoft Visual Basic.

Information Catalog Center Tutorial
Create and manage an information catalog to locate and use metadata using the Information Catalog Center.

Video Central for e-business Tutorial
Develop and deploy an advanced DB2 Web Services application using WebSphere® products.

Visual Explain Tutorial
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 Information Center for topics

The DB2® Information Center gives you access to all of the information you need to take full advantage of DB2 Universal Database™ and DB2 Connect™ in your business. The DB2 Information Center also documents major DB2 features and components including replication, data warehousing, the Information Catalog Center, Life Sciences Data Connect, and DB2 extenders.

The DB2 Information Center accessed from a browser has the following features:

Regularly updated documentation
Keep your topics up-to-date by downloading updated HTML.

Search

Search all of the topics installed on your workstation by clicking **Search** in the navigation toolbar.

Integrated navigation tree

Locate any topic in the DB2 library from a single navigation tree. The navigation tree is organized by information type as follows:

- Tasks provide step-by-step instructions on how to complete a goal.
- Concepts provide an overview of a subject.
- Reference topics provide detailed information about a subject, including statement and command syntax, message help, requirements.

Master index

Access the information in topics and tools help from one master index. The index is organized in alphabetical order by index term.

Master glossary

The master glossary defines terms used in the DB2 Information Center. The glossary is organized in alphabetical order by glossary term.

Related tasks:

- "Finding topics by accessing the DB2 Information Center from a browser" on page 812
- "Finding product information by accessing the DB2 Information Center from the administration tools" on page 814
- "Updating the HTML documentation installed on your machine" on page 816

Appendix B. Notices

IBM may not offer the products, services, or features discussed in this document in all 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/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country/region 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 that 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 document 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 may contain 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 may contain 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.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both, and have been used in at least one of the documents in the DB2 UDB documentation library.

ACF/VTAM
AISPO
AIX
AIXwindows
AnyNet
APPN
AS/400
BookManager
C Set++
C/370
CICS
Database 2
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
DB2 Connect
DB2 Extenders
DB2 OLAP Server
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
eServer
Extended Services
FFST
First Failure Support Technology
IBM
IMS
IMS/ESA
iSeries
LAN Distance
MVS
MVS/ESA
MVS/XA
Net.Data
NetView
OS/390
OS/400
PowerPC
pSeries
QBIC
QMF
RACF
RISC System/6000
RS/6000
S/370
SP
SQL/400
SQL/DS
System/370
System/390
SystemView
Tivoli
VisualAge
VM/ESA
VSE/ESA
VTAM
WebExplorer
WebSphere
WIN-OS/2
z/OS
zSeries

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 UDB documentation library:

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

V

W

Contacting IBM

In the United States, call one of the following numbers to contact IBM:

- 1-800-237-5511 for customer service
- 1-888-426-4343 to learn about available service options
- 1-800-IBM-4YOU (426-4968) for DB2 marketing and sales

In Canada, call one of the following numbers to contact IBM:

- 1-800-IBM-SERV (1-800-426-7378) for customer service
- 1-800-465-9600 to learn about available service options
- 1-800-IBM-4YOU (1-800-426-4968) for DB2 marketing and sales

To locate an IBM office in your country or region, check IBM's Directory of Worldwide Contacts on the web at www.ibm.com/planetwide

Product information

Information regarding DB2 Universal Database products is available by telephone or by the World Wide Web at www.ibm.com/software/data/db2/udb

This site contains the latest information on the technical library, ordering books, client downloads, newsgroups, FixPaks, news, and links to web resources.

If you live in the U.S.A., then you can call one of the following numbers:

- 1-800-IBM-CALL (1-800-426-2255) to order products or to obtain general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, go to the IBM Worldwide page at www.ibm.com/planetwide

Part Number: CT17SNA

Printed in U.S.A.

(1P) P/N: CT17SNA

SC09-4845-00

Spine information:

IBM® DB2 Universal Database™ SQL Reference, Volume 2 Version 8