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 may not 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 as described in "Basic Predicate" on page 187.

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. The supported SQL is currently limited to the RETURN statement.

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). Nicknames or OLEDB table functions 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 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.

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
See "CREATE FUNCTION (External Scalar)" on page 590 for details on predicate specifications.

RETURN
Specifies the return value of the function. Parameter names can be referenced in the RETURN statement. Parameter names may be qualified by the function name to avoid ambiguous references.

expression
Specifies the expression to be returned for the function. The result data type of the expression must be assignable (using store assignment rules) to the data type defined in the RETURNS clause (SQLSTATE 42866). A scalar expression (other than a scalar fullselect) cannot be specified for a table function (SQLSTATE 428F1).

NULL
Specifies that the function returns a null value of the data type defined in the RETURNS clause.

WITH *common-table-expression*
Defines a common table expression for use with the fullselect that follows. See "common-table-expression" on page 440.

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), and 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).

If the function is a scalar function, the fullselect must return one column (SQLSTATE 42823) and, at most, one row (SQLSTATE 21000).

If the function is a row function, it must return, at most, one row (SQLSTATE 21505).

If the function is a table function, it can return zero or more rows with one or more columns.

Notes

- 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 must not contain a recursive call to itself or to another function or method that calls it.
- 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.
  - Since 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.

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
```

Note that the definer of this function must have the SELECT WITH GRANT OPTION privilege on the EMPLOYEE table and that all users may invoke the table function DEPTEMPLOYEES, effectively giving them access to the data in the result columns for each department number.

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.

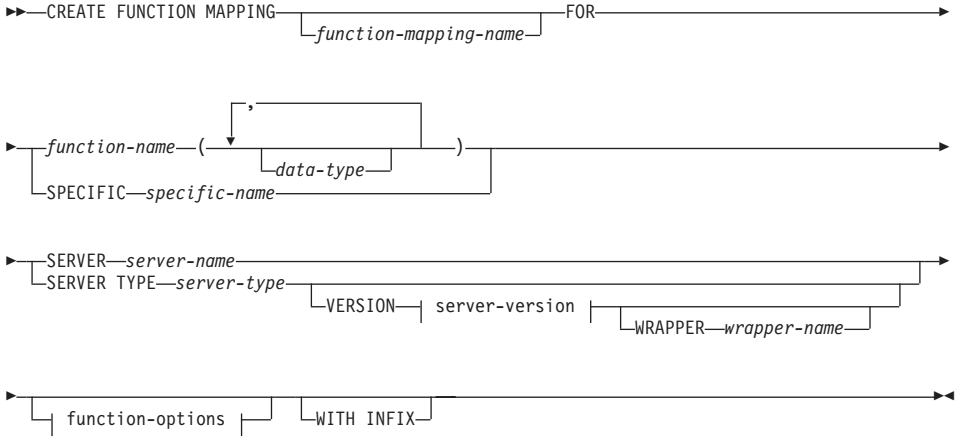
Invocation

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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

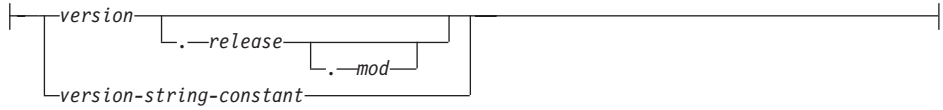
Authorization

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

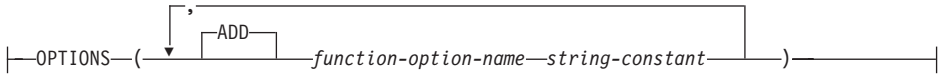
Syntax



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. Refer to "Function Mapping Options" on page 1248 for descriptions of *function-option-name*s and their settings.

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' )
```

CREATE INDEX

The CREATE INDEX statement is used to create:

- An index on a DB2 table
- 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

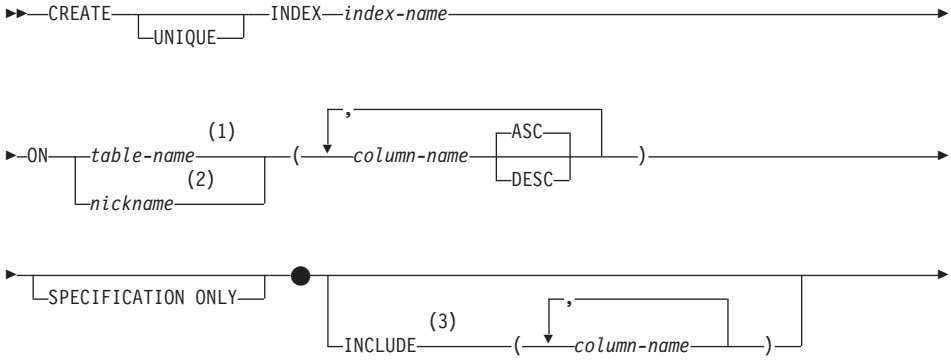
Authorization

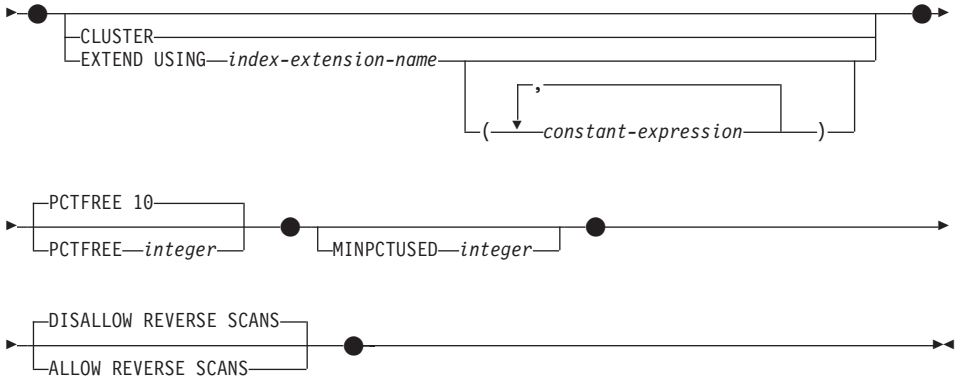
The privileges held by the authorization ID of the statement must include at least one of the following:

- 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.

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 will create an index specification. INCLUDE, CLUSTER, PCTFREE, MINPCTUSED, DISALLOW REVERSE SCANS, and ALLOW REVERSE SCANS 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. For details refer to "Appendix J. Interaction of Triggers and Constraints" on page 1287.

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).

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.

The table-name cannot be a declared temporary table (SQLSTATE 42995).

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. The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939)

ON *table-name* **or** *nickname*
The *table-name* names a table on which an index is to be created. The table must be a base table (not a view) or a summary table described in the catalog. It must not name a catalog table (SQLSTATE 42832), or a declared temporary table (SQLSTATE 42995). 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 summary 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. 16 columns or less may be specified. If *table-name* is a typed table, 15 columns or less may 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* may 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. See "Byte Counts" on page 757 for more information on overhead affecting this limit.

The length of any individual column must not be greater than 255 bytes. 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 255 byte limit

(SQLSTATE 42962). 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).

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.

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 (42601).

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).

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).

MINPCTUSED *integer*
Indicates whether indexes are reorganized online and the threshold for the minimum percentage of space used on an index leaf page If after a key is deleted from an index leaf page, the percentage of space used on the page is at or below *integer* percentage, 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.

MINPCTUSED is disallowed if *nickname* is specified (SQLSTATE 42601).

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).

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

- 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. (See *Command Reference* for information about RUNSTATS.) 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.
- 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. Refer to "SET CURRENT EXPLAIN MODE" on page 1006 for more details.
- 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.
- For more information about index specifications, see "Index Specifications" on page 49.

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. For a definition of index extensions, please see "CREATE INDEX EXTENSION" on page 669.

```
CREATE INDEX SPATIAL_INDEX ON CUSTOMER (LOCATION)
     EXTEND USING (GRID_EXTENSION (x'000100100010001000400010'))
```

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

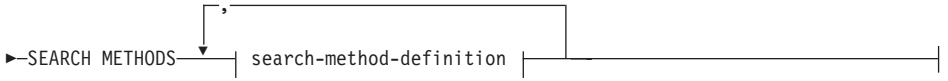
This statement can be embedded in an application program or issued through dynamic SQL statements. It is an executable statement that can be dynamically prepared. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

Authorization

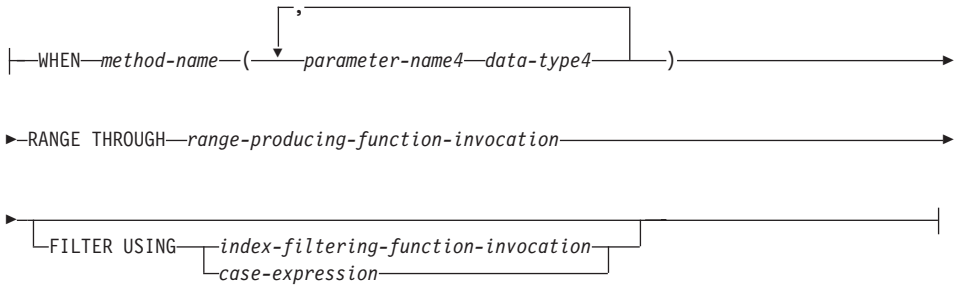
The privileges held by the authorization ID of the statement must include at least one of the following:

- 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



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). See "Constants" on page 115 for more information about constants. 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:

1. Resolve to a table function (SQLSTATE 428E4)
2. Not be defined with LANGUAGE SQL (SQLSTATE 428E4)
3. Not be defined with NOT DETERMINISTIC (SQLSTATE 428E4) or EXTERNAL ACTION (SQLSTATE 428E4)

4. 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.
5. Not include a subquery (SQLSTATE 428E3).
6. 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).

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:

1. Resolve to a table function (SQLSTATE 428E4)
2. Not include a subquery (SQLSTATE 428E3) or SQL function (SQLSTATE 428E4) in its arguments
3. Not be defined with LANGUAGE SQL (SQLSTATE 428E4)
4. Not be defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 428E4)
5. 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.

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:

1. Not be defined with LANGUAGE SQL (SQLSTATE 429B4)
2. Not be defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 42845)
3. Not have a structured data type in the data type of any of the parameters (SQLSTATE 428E3).
4. 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.

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
```

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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

Authorization

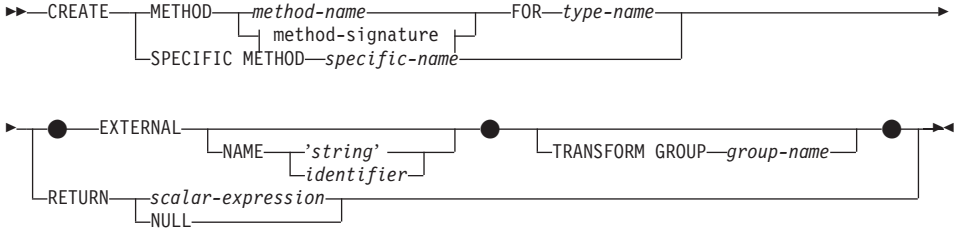
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATEIN privilege on the schema of the structured type referred to in CREATE METHOD
- The DEFINER of the structured type referred to in the CREATE METHOD statement.

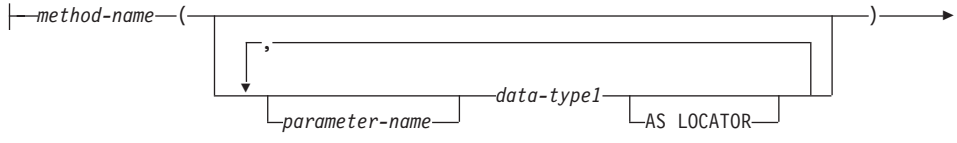
If the authorization ID of the statement does not have SYSADM or DBADM authority, and the method identifies a table or view in the RETURN statement, the privileges that the authorization ID of the statement holds (without considering group privileges) must include SELECT WITH GRANT OPTION for each identified table and view.

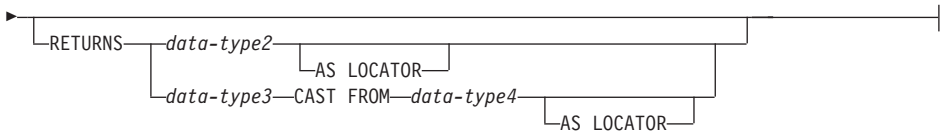
If the authorization ID has insufficient authority to perform the operation, an error is raised (SQLSTATE 42502).

Syntax



method-signature:





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. See "CREATE FUNCTION (External Scalar)" on page 590 for more information of the specific language conventions.

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).

RETURN *scalar-expression* **or NULL**
The RETURN statement is an SQL control statement that specifies the value returned by the method.

scalar-expression
An expression that specifies the body of the method when the method-specification on CREATE TYPE specifies LANGUAGE SQL. Parameter names can be referenced in the expression. The subject of the method is passed to the method implementation in the form of an implicit first parameter named SELF. The result data type of the expression must be assignable (using store assignment rules) to the data type defined in the RETURNS clause of the method-specification on CREATE TYPE (SQLSTATE 42866).

The expression must comply with the following parts of the method-specification:

- DETERMINISTIC or NOT DETERMINISTIC (SQLSTATE 428C2)

- EXTERNAL ACTION or NO EXTERNAL ACTION (SQLSTATE 428C2)
- CONTAINS SQL or READS SQL DATA (SQLSTATE 42985)

NULL
Specifies that the function returns a null value. The null value is of the data type defined in the RETURNS clause of the method-specification created with the CREATE TYPE 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).

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
```

CREATE NICKNAME

The CREATE NICKNAME statement creates a nickname for a data source table or 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

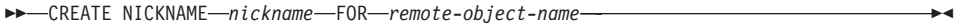
Authorization

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



Description

nickname
Names the federated server's identifier for the table or view that is referenced by *remote-object-name*. 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
Names a three-part identifier with this format:

data-source-name.remote-schema-name.remote-table-name

where:

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.

remote-table-name
Names either of the following identifiers:

- The name or an alias of a DB2 family table or view
- The name of an Oracle table or view
- The *table-name* cannot be a declared temporary table (SQLSTATE 42995)

Notes

- The table or 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, large object (LOB) types, 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. Refer to the *Application Development Guide* for details.
- 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 table or view, DB2 stores the names of the table's or view's columns in the catalog. If a name exceeds the

maximum allowable length for DB2 column names (30 characters), DB2 truncates the name to this length. If the truncated version is not unique among the other names of the table's or view's columns, DB2 makes it unique by following the procedure described in the preceding paragraph.

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 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
```

CREATE NODEGROUP

The CREATE NODEGROUP statement creates a new nodegroup within the database and assigns partitions or nodes to the nodegroup, and records the nodegroup definition in the catalog.

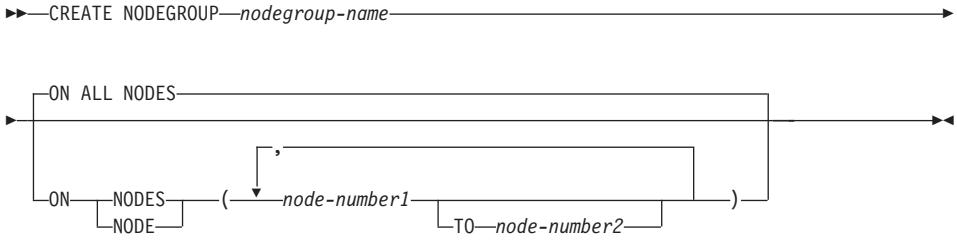
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be prepared dynamically. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

Authorization

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

Syntax



Description

nodegroup-name
Names the nodegroup. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *nodegroup-name* must not identify a nodegroup that already exists in the catalog (SQLSTATE 42710). The *nodegroup-name* must not begin with the characters ″SYS″ or ″IBM″ (SQLSTATE 42939).

ON ALL NODES
Specifies that the nodegroup is defined over all partitions defined to the database (`db2nodes.cfg` file) at the time the nodegroup is created.

If a partition is added to the database system, the ALTER NODEGROUP statement should be issued to include this new partition in a nodegroup (including IBMDEFAULTGROUP). Furthermore, the REDISTRIBUTE NODEGROUP command must be issued to move data to the partition. Refer to the *Administrative API Reference* or the *Command Reference* for more information.

ON NODES
Specifies the specific partitions that are in the nodegroup. NODE is a synonym for NODES.

node-number1
Specify a specific partition number. [72]

TO *node-number2*
Specify a range of partition numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers are included in the nodegroup.

Rules

- Each partition or node specified by number must be defined in the `db2nodes.cfg` file (SQLSTATE 42729).
- Each *node-number* listed in the ON NODES clause must be appear at most once (SQLSTATE 42728).
- A valid *node-number* is between 0 and 999 inclusive (SQLSTATE 42729).

Notes

- This statement creates a partitioning map for the nodegroup (Refer to "Data Partitioning Across Multiple Partitions" on page 59 for more information) . 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.NODEGROUPS 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 nodegroup, the corresponding partitioning map has only one entry. For a multiple partition nodegroup, 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.

Example

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 nodegroup call MAXGROUP on all six partitions. The statement is as follows:

```
CREATE NODEGROUP MAXGROUP
ON ALL NODES
```

- Assume that you want to create a nodegroup MEDGROUP on partitions 0, 1, 2, 5, 8. The statement is as follows:

```
CREATE NODEGROUP MEDGROUP
 ON NODES (0 TO 2, 5, 8)
```

72. node-name of the form 'NODEnnnnn' may be specified for compatibility with the previous version.

- Assume that you want to create a single-partition nodegroup MINGROUP on partition (or node) 7. The statement is as follows:

```
CREATE NODEGROUP MINGROUP
 ON NODE (7)
```

Note: The singular form of the keyword NODES is also accepted.