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. See "CREATE FUNCTION (External Scalar)" on page 590.

- 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. See "CREATE FUNCTION (External Table)" on page 615.

- 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. See "CREATE FUNCTION (OLE DB External Table)" on page 631.

- Source 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. See "CREATE FUNCTION (Source or Template)" on page 639.

  It is possible to create a partial function, called a *function template*, that defines what types of values are to be returned but 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. See "CREATE FUNCTION (SQL Scalar, Table or Row)" on page 649.

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. 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 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.

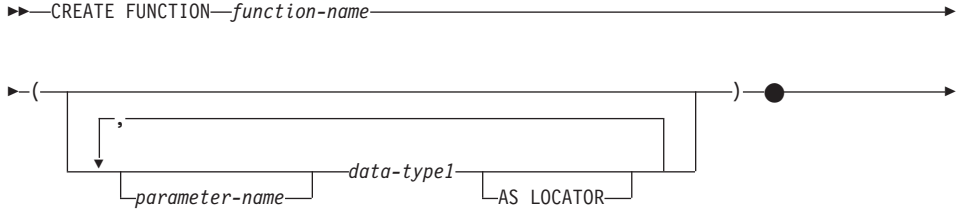
To create a not-fenced function, the privileges held by the authorization ID of the statement must also include at least one of the following:

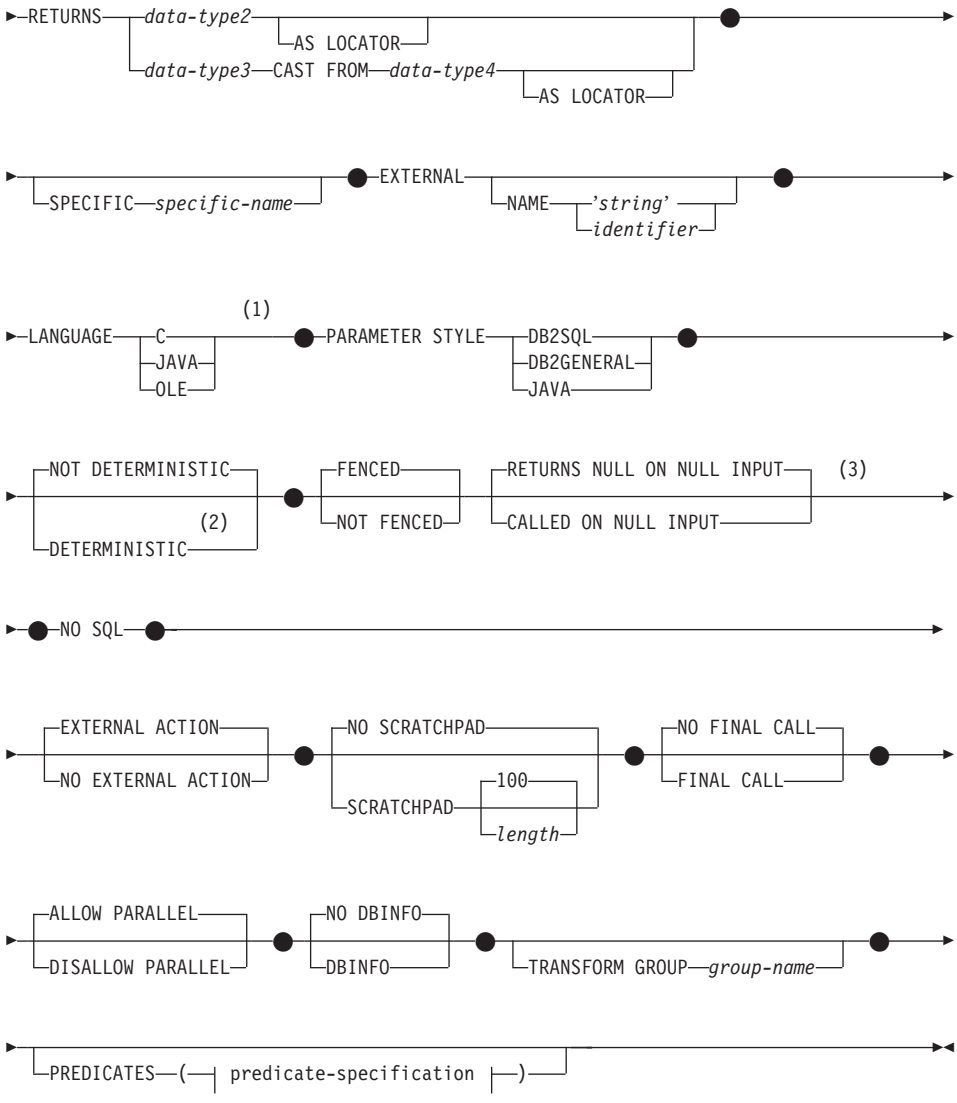
- CREATE_NOT_FENCED authority on the database
- SYSADM or DBADM authority.

To create a fenced function, no additional authorities or privileges are required.

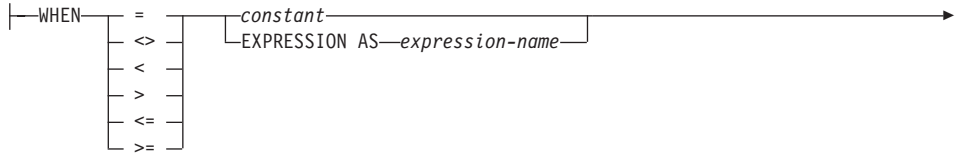
If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

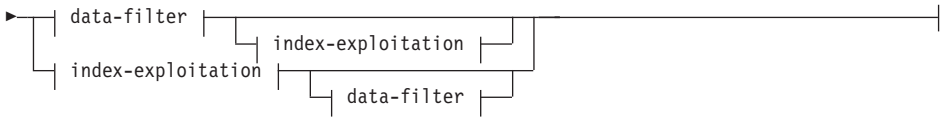
Syntax



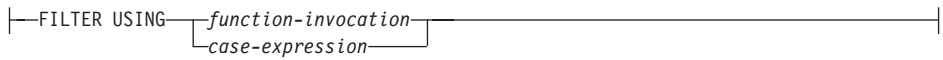


predicate-specification:

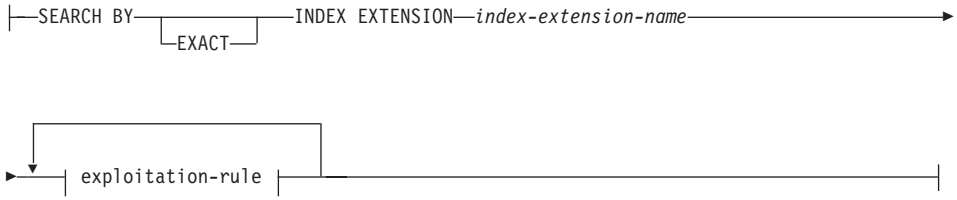




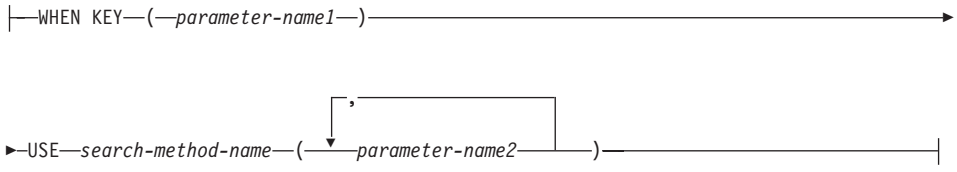
data-filter:



index-exploitation:



exploitation-rule:



Notes:

1 LANGUAGE SQL is also supported. See "CREATE FUNCTION (SQL Scalar, Table or Row)" on page 649.

2 NOT VARIANT may be specified in place of DETERMINISTIC and VARIANT may be specified in place of NOT DETERMINISTIC.

3 NULL CALL may be specified in place of CALLED ON NULL INPUT and NOT NULL CALL may be specified in place of RETURNS NULL ON NULL INPUT.

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 may not 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 as described in "Basic Predicate" on page 187. 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). 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. See the language-specific sections of the *Application Development Guide* for details on the mapping between the SQL data types and host language data types with respect to user-defined functions.
- DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815). For alternatives to using DECIMAL refer to *Application Development Guide*.
- 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. Use of LOB locators in UDFs are described in *Application Development Guide*.

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, 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 (for the definition of castable, see "Casting Between Data Types" on page 91).

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. Use of LOB locators in UDFs are described in *Application Development Guide*.

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.

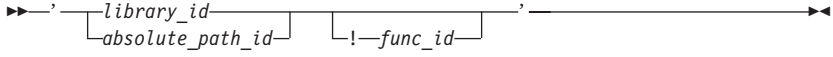
If NAME clause is not specified "NAME *function-name*" is assumed.

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 raised.



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 (OS/2, and 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.

'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 OS/2, and 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.In OS/2 the library_id should not contain more than 8 characters.

absolute_path_id
Identifies the full path name of the file containing 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 shared library.

In OS/2, and Windows 32-bit operating systems, 'd:\mylib\myfunc' would cause the database manager to load dynamic link library, myfunc.dll file, from the d:\mylib directory.In OS/2 the last part of this specification (i.e. the name of the dll), should not contain more than 8 characters.

! 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.

In OS/2, and 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).

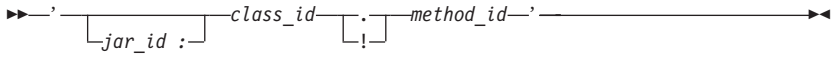
If the string is not properly formed, an error (SQLSTATE 42878) is raised.

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.

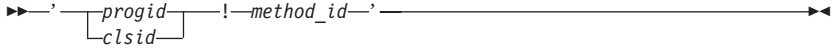
jar_id
Identifies the jar identifier given to the jar collection when it was installed in the database. It can be either a simple identifier, or a schema qualified identifier. Examples are 'myJar' and 'mySchema.myJar'.

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. In OS/2 and 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 *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.

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

DB2SQL
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.

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.

Refer to *Application Development Guide* for details on passing parameters.

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 (NOT FENCED), or not (FENCED).

If a function is registered as FENCED, the database manager insulates its internal resources (e.g. 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.

Warning: 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.

Note that, while the use of FENCED does offer a greater degree of protection for database integrity than NOT FENCED, a FENCED UDF that has not been adequately coded, reviewed and tested can also cause an inadvertent failure of DB2.

Most user-defined functions should be able to run either as FENCED or NOT FENCED. Only FENCED can be specified for a function with LANGUAGE OLE (SQLSTATE 42613).

If the function is FENCED, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

To change from FENCED to NOT FENCED, the function must be re-registered (by first dropping it and then re-creating it). Either SYSADM authority, DBADM authority or a special authority (CREATE_NOT_FENCED) is required to register a user-defined function as NOT FENCED.

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
This mandatory clauses indicates that the function cannot issue any SQL statements. If it does, an error (SQLSTATE 38502) is raised at run time.

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.

NO FINAL CALL or **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-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 function, and no final call is made.

A description of the scalar UDF processing of these calls when errors occur is included in the *Application Development Guide*.

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 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 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.

Please see the *Application Development Guide* for detailed information on the structure and how it is passed to the user-defined function.

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.

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

- 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" on page 90). 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)" on page 792.
- 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).
- For information on writing, compiling, and linking an external user-defined function, see the *Application Development Guide*.
- 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.

Examples

Example 1: Pellow is registering the CENTRE function in his PELLOW schema. Let those keywords that will default default, and let the system provide a function specific name:

```
CREATE FUNCTION CENTRE (INT,FLOAT)
    RETURNS FLOAT
    EXTERNAL NAME 'mod!middle'
    LANGUAGE C
    PARAMETER STYLE DB2SQL
    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 DB2SQL
    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 the null checking), if the CREATE FUNCTION statement had used NOT NULL CALL. Further examples of user-defined function programs can be found in the *Application Development Guide*. The CREATE FUNCTION statement:

```
CREATE FUNCTION ntest1 (SMALLINT)
   RETURNS SMALLINT
   EXTERNAL NAME 'ntest1!nudft1'
   LANGUAGE C    PARAMETER STYLE DB2SQL
   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 DB2SQL
     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 DB2SQL
     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.

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. 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 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 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