CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger in the database.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared. 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 when the trigger is created must include at least one of the following:

- SYSADM or DBADM authority.
- ALTER privilege on the table on which the trigger is defined, or ALTERIN privilege on the schema of the table on which the trigger is defined and one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the trigger does not exist
  - CREATEIN privilege on the schema, if the schema name of the trigger refers to an existing schema.

If the authorization ID of the statement does not have SYSADM or DBADM authority, the privileges that the authorization ID of the statement holds (without considering PUBLIC or group privileges) must include all of the following as long as the trigger exists:

- SELECT privilege on the table on which the trigger is defined, if any transition variables or tables are specified
- SELECT privilege on any table or view referenced in the triggered action condition
- Necessary privileges to invoke the triggered SQL statements specified.

If a trigger definer can only create the trigger because the definer has SYSADM authority, then the definer is granted explicit DBADM authority for the purpose of creating the trigger.

Syntax

```
►►──CREATE TRIGGER──trigger-name───┬─NO CASCADE BEFORE─┬──────────────────────────────►
                                   └─AFTER─────────────┘
```

```
►──┬─INSERT─────────────────────────┬─ON──table-name──────────────────►
   ├─DELETE─────────────────────────┤
   └─UPDATE──┬────────────────────┬─┘
             │    ┌─,──────────┐  │
             │    ▼            │  │
             └─OF───column-name─┴──┘

►──┬──────────────────────────────────────────────────────────────┬──►
   │            ┌───────────────────────────────────────────────┐ │
   │            ▼   (1)   (2)              ┌─AS─┐                │ │
   └─REFERENCING─────────────────┬─OLD─────┴────┴─correlation-name─┬┴─┘
                                 │        ┌─AS─┐                   │
                                 ├─NEW────┴────┴─correlation-name──┤
                                 │            ┌─AS─┐               │
                                 ├─OLD_TABLE──┴────┴─identifier────┤
                                 │            ┌─AS─┐               │
                                 └─NEW_TABLE──┴────┴─identifier────┘

►──┬─FOR EACH ROW─────────────┬─MODE DB2SQL──┤ triggered-action ├──────►◄
   │ (3)                      │
   └──────FOR EACH STATEMENT──┘
```

triggered-action:

```
├──┬────────────────────────────────────┬─────────────────────────────►
   └─WHEN──(──search-condition──)───────┘

►──┬─triggered-SQL-statement───────────────────────────────────┬──────┤
   │               ┌─────────────────────────────────┐          │
   │               ▼                                 │          │
   └─BEGIN ATOMIC────triggered-SQL-statement──;──────┴──END─────┘
```

Notes:

1 OLD and NEW may only be specified once each.

2 OLD_TABLE and NEW_TABLE may only be specified once each and only for AFTER triggers.

3 FOR EACH STATEMENT may not be specified for BEFORE triggers.

Description

trigger-name
: Names the trigger. The name, including the implicit or explicit schema name must not identify a trigger already described in the catalog (SQLSTATE 42710). If a two part name is specified, the schema name cannot begin with ″SYS″ (SQLSTATE 42939).

NO CASCADE BEFORE
Specifies that the associated triggered action is to be applied before any changes caused by the actual update of the subject table are applied to the database. It also specifies that the triggered action of the trigger will not cause other triggers to be activated.

AFTER
Specifies that the associated triggered action is to be applied after the changes caused by the actual update of the subject table are applied to the database.

INSERT
Specifies that the triggered action associated with the trigger is to be executed whenever an INSERT operation is applied to the designated base table.

DELETE
Specifies that the triggered action associated with the trigger is to be executed whenever a DELETE operation is applied to the designated base table.

UPDATE
Specifies that the triggered action associated with the trigger is to be executed whenever an UPDATE operation is applied to the designated base table subject to the columns specified or implied.

If the optional *column-name* list is not specified, every column of the table is implied. Therefore, omission of the *column-name* list implies that the trigger will be activated by the update of any column of the table.

OF *column-name***,...**
Each *column-name* specified must be a column of the base table (SQLSTATE 42703). If the trigger is a BEFORE trigger, the *column-name* specified may not be a generated column other than the identity column (SQLSTATE 42989). No *column-name* shall appear more than once in the *column-name* list (SQLSTATE 42711). The trigger will only be activated by the update of a column identified in the *column-name* list.

ON *table-name*
Designates the subject table of the trigger definition. The name must specify a base table or an alias that resolves to a base table (SQLSTATE 42809). The name must not specify a catalog table (SQLSTATE 42832), a summary table (SQLSTATE 42997), a declared temporary table (SQLSTATE 42995), or a nickname (SQLSTATE 42809).

REFERENCING
Specifies the correlation names for the *transition variables* and the table names for the *transition tables*. Correlation names identify a specific row in the set of rows affected by the triggering SQL operation. Table names

identify the complete set of affected rows. Each row affected by the triggering SQL operation is available to the triggered action by qualifying columns with *correlation-names* specified as follows.

OLD AS *correlation-name*
Specifies a correlation name which identifies the row state prior to the triggering SQL operation.

NEW AS *correlation-name*
Specifies a correlation name which identifies the row state as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed.

The complete set of rows affected by the triggering SQL operation is available to the triggered action by using a temporary table name specified as follows.

OLD_TABLE AS *identifier*
Specifies a temporary table name which identifies the set of affected rows prior to the triggering SQL operation.

NEW_TABLE AS *identifier*
Specifies a temporary table name which identifies the affected rows as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed.

The following rules apply to the REFERENCING clause:

- None of the OLD and NEW correlation names and the OLD_TABLE and NEW_TABLE names can be identical (SQLSTATE 42712).
- Only one OLD and one NEW *correlation-name* may be specified for a trigger (SQLSTATE 42613).
- Only one OLD_TABLE and one NEW_TABLE *identifier* may be specified for a trigger (SQLSTATE 42613).
- The OLD *correlation-name* and the OLD_TABLE *identifier* can only be used if the trigger event is either a DELETE operation or an UPDATE operation (SQLSTATE 42898). If the operation is a DELETE operation, OLD *correlation-name* captures the value of the deleted row. If it is an UPDATE operation, it captures the value of the row before the UPDATE operation. The same applies to the OLD_TABLE *identifier* and the set of affected rows.
- The NEW *correlation-name* and the NEW_TABLE *identifier* can only be used if the trigger event is either an INSERT operation or an UPDATE operation (SQLSTATE 42898). In both operations, the value of NEW captures the new state of the row as provided by the original operation and as modified by any BEFORE trigger that has executed to this point. The same applies to the NEW_TABLE *identifier* and the set of affected rows.

- OLD_TABLE and NEW_TABLE *identifier*s cannot be defined for a BEFORE trigger (SQLSTATE 42898).
- OLD and NEW *correlation-name*s cannot be defined for a FOR EACH STATEMENT trigger (SQLSTATE 42899).
- Transition tables cannot be modified (SQLSTATE 42807).
- The total of the references to the transition table columns and transition variables in the triggered-action cannot exceed the limit for the number of columns in a table or the sum of their lengths cannot exceed the maximum length of a row in a table (SQLSTATE 54040).
- The scope of each *correlation-name* and each *identifier* is the entire trigger definition.

FOR EACH ROW
Specifies that the triggered action is to be applied once for each row of the subject table that is affected by the triggering SQL operation.

FOR EACH STATEMENT
Specifies that the triggered action is to be applied only once for the whole statement. This type of trigger granularity cannot be specified for a BEFORE trigger (SQLSTATE 42613). If specified, an UPDATE or DELETE trigger is activated even when no rows are affected by the triggering UPDATE or DELETE statement.

MODE DB2SQL
This clause is used to specify the mode of triggers. This is the only valid mode currently supported.

triggered-action
Specifies the action to be performed when a trigger is activated. A triggered-action is composed of one or several *triggered-SQL-statement*s and by an optional condition for the execution of the *triggered-SQL-statement*s. If there is more than one *triggered-SQL-statement* in the triggered-action for a given trigger, they must be enclosed within the BEGIN ATOMIC and END keywords, separated by a semi-colon, [83] and are executed in the order they are specified.

WHEN (*search-condition***)**
Specifies a condition that is true, false, or unknown. The *search-condition* provides a capability to determine whether or not a certain triggered action should be executed.

The associated action is performed only if the specified search condition evaluates as true. If the WHEN clause is omitted, the associated *triggered-SQL-statement*s are always performed.

83. When using this form in the Command Line Processor, the statement terminating character cannot be the semi-colon. See the *Command Reference* for information on specifying an alternative terminating character.

triggered-SQL-statement

If the trigger is a BEFORE trigger, then a triggered SQL statement must be one of the following (SQLSTATE 42987):

- a fullselect [84]
- a SET transition-variable SQL statement.
- a SIGNAL SQLSTATE statement

If the trigger is an AFTER trigger, then a triggered SQL statement must be one of the following (SQLSTATE 42987):

- an INSERT SQL statement
- a searched UPDATE SQL statement
- a searched DELETE SQL statement
- a SIGNAL SQLSTATE statement
- a fullselect [84]

The *triggered-SQL-statement* cannot reference an undefined transition variable (SQLSTATE 42703) or a declared temporary table (SQLSTATE 42995).

The *triggered-SQL-statement* in a BEFORE trigger cannot reference a summary table defined with REFRESH IMMEDIATE (SQLSTATE 42997).

The *triggered-SQL-statement* in a BEFORE trigger cannot reference a generated column, other than the identity column, in the new transition variable (SQLSTATE 42989).

Notes

- Adding a trigger to a table that already has rows in it will not cause any triggered actions to be activated. Thus, if the trigger is designed to enforce constraints on the data in the table, those constraints may not be satisfied by the existing rows.
- If the events for two triggers occur simultaneously (for example, if they have the same event, activation time, and subject tables), then the first trigger created is the first to execute.
- If a column is added to the subject table after triggers have been defined, the following rules apply:
  - If the trigger is an UPDATE trigger that was specified without an explicit column list, then an update to the new column will cause the activation of the trigger.
  - The column will not be visible in the triggered action of any previously defined trigger.

84. A common-table-expression may precede a fullselect.

  - The OLD_TABLE and NEW_TABLE transition tables will not contain this column. Thus, the result of performing a ″SELECT *″ on a transition table will not contain the added column.
- If a column is added to any table referenced in a triggered action, the new column will not be visible to the triggered action.
- The result of a fullselect specified as a *triggered-SQL-statement* is not available inside or outside of the trigger.
- A before delete trigger defined on a table involved in a cycle of cascaded referential constraints should not include references to the table on which it is defined or any other table modified by cascading during the evaluation of the cycle of referential integrity constraints. The results of such a trigger are data dependent and therefore may not produce consistent results.

  In its simplest form, this means that a before delete trigger on a table with a self-referencing referential constraint and a delete rule of CASCADE should not include any references to the table in the *triggered-action*.
- The creation of a trigger causes certain packages to be marked invalid:
  - If an update trigger without an explicit column list is created, then packages with an update usage on the target table are invalidated.
  - If an update trigger with a column list is created, then packages with update usage on the target table are only invalidated if the package also has an update usage on at least one column in the *column-name* list of the CREATE TRIGGER statement.
  - If an insert trigger is created, packages that have an insert usage on the target table are invalidated.
  - If a delete trigger is created, packages that have a delete usage on the target table are invalidated.
- A package remains invalid until the application program is explicitly bound or rebound, or it is executed and the database manager automatically rebinds it.
- ***Inoperative triggers***: An *inoperative trigger* is a trigger that is no longer available and is therefore never activated. A trigger becomes inoperative if:
  - A privilege that the creator of the trigger is required to have for the trigger to execute is revoked.
  - An object such as a table, view or alias, upon which the triggered action is dependent, is dropped.
  - A view, upon which the triggered action is dependent, becomes inoperative.
  - An alias that is the subject table of the trigger is dropped.

  In practical terms, an inoperative trigger is one in which a trigger definition has been dropped as a result of cascading rules for DROP or REVOKE statements. For example, when an view is dropped, any trigger with a *triggered-SQL-statement* defined using that view is made inoperative.

When a trigger is made inoperative, all packages with statements performing operations that were activating the trigger will be marked invalid. When the package is rebound (explicitly or implicitly) the **inoperative trigger is completely ignored**. Similarly, applications with dynamic SQL statements performing operations that were activating the trigger will also completely ignore any inoperative triggers.

The trigger name can still be specified in the DROP TRIGGER and COMMENT ON TRIGGER statements.

An inoperative trigger may be recreated by issuing a CREATE TRIGGER statement using the definition text of the inoperative trigger. This trigger definition text is stored in the TEXT column of SYSCAT.TRIGGERS. Note that there is no need to explicitly drop the inoperative trigger in order to recreate it. Issuing a CREATE TRIGGER statement with the same *trigger-name* as an inoperative trigger will cause that inoperative trigger to be replaced with a warning (SQLSTATE 01595).

Inoperative triggers are indicated by an X in the VALID column of the SYSCAT.TRIGGERS catalog view.

- ***Errors executing triggers***: Errors that occur during the execution of triggered-SQL-statements are returned using SQLSTATE 09000 unless the error is considered severe. If the error is severe, the severe error SQLSTATE is returned. The SQLERRMC field of the SQLCA for non-severe error will include the trigger name, SQLCODE, SQLSTATE and as many tokens as will fit from the tokens of the failure.

  A *triggered-SQL-statement* could be a SIGNAL SQLSTATE statement or contain a RAISE_ERROR function. In both these cases, the SQLSTATE returned is the one specified in the SIGNAL SQLSTATE statement or the RAISE_ERROR condition.
- Creating a trigger with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- A value generated by the database manager for an identity column is generated before the execution of any BEFORE triggers. Therefore, the generated identity value is visible to BEFORE triggers.
- A value generated by the database manager for a generated by expression column is generated after the execution of all BEFORE triggers.Therefore, the value generated by the expression is not visible to BEFORE triggers.
- ***Triggers and typed tables***: A trigger can be attached to a typed table at any level of a table hierarchy. If an SQL statement activates multiple triggers, the triggers will be executed in their creation order, even if they are attached to different tables in the typed table hierarchy.

When a trigger is activated, its transition variables (OLD, NEW, OLD_TABLE and NEW_TABLE) may contain rows of subtables. However, they will contain only columns defined on the table to which they are attached.

Effects of INSERT, UPDATE, and DELETE statements:

– Row triggers: When an SQL statement is used to INSERT, UPDATE, or DELETE a table row, it activates row-triggers attached to the most specific table containing the row, and all supertables of that table. This rule is always true, regardless of how the SQL statement accesses the table. For example, when issuing an UPDATE EMP command, some of the updated rows may be in the subtable MGR. For EMP rows, the row-triggers attached to EMP and its supertables are activated. For MGR rows, the row-triggers attached to MGR and its supertables are activated.

– Statement triggers: An INSERT, UPDATE, or DELETE statement activates statement-triggers attached to tables (and their supertables) that could be affected by the statement. This rule is always true, regardless of whether any actual rows in these tables were affected. For example, on an INSERT INTO EMP command, statement-triggers for EMP and its supertables are activated. As another example, on either an UPDATE EMP or DELETE EMP command, statement triggers for EMP and its supertables and subtables are activated, even if no subtable rows were updated or deleted. Likewise, a UPDATE ONLY (EMP) or DELETE ONLY (EMP) command will activate statement-triggers for EMP and its supertables, but not statement-triggers for subtables.

Effects of DROP TABLE statements: A DROP TABLE statement does not activate any triggers that are attached to the table being dropped. However, if the dropped table is a subtable, all the rows of the dropped table are considered to be deleted from its supertables. Therefore, for a table T:

– Row triggers: DROP TABLE T activates row-type delete-triggers that are attached to all supertables of T, for each row of T.

– Statement triggers: DROP TABLE T activates statement-type delete-triggers that are attached to all supertables of T, regardless of whether T contains any rows.

Actions on Views: To predict what triggers are activated by an action on a view, use the view definition to translate that action into an action on base tables. For example:

1. An SQL statement performs UPDATE V1, where V1 is a typed view with a subview V2. Suppose V1 has underlying table T1, and V2 has underlying table T2. The statement could potentially affect rows in T1, T2, and their subtables, so statement triggers are activated for T1 and T2 and all their subtables and supertables.
2. An SQL statement performs UPDATE V1, where V1 is a typed view with a subview V2. Suppose V1 is defined as SELECT ... FROM

ONLY(T1) and V2 is defined as SELECT ... FROM ONLY(T2). Since the statement cannot affect rows in subtables of T1 and T2, statement triggers are activated for T1 and T2 and their supertables, but not their subtables.

3. An SQL statement performs UPDATE ONLY(V1), where V1 is a typed view defined as SELECT ... FROM T1. The statement can potentially affect T1 and its subtables. Therefore, statement triggers are activated for T1 and all its subtables and supertables.
4. An SQL statement performs UPDATE ONLY(V1), where V1 is a typed view defined as SELECT ... FROM ONLY(T1). In this case, T1 is the only table that can be affected by the statement, even if V1 has subviews and T1 has subtables. Therefore, statement triggers are activated only for T1 and its supertables.

Examples

Example 1: Create two triggers that will result in the automatic tracking of the number of employees a company manages. The triggers will interact with the following tables:

EMPLOYEE table with these columns: ID, NAME, ADDRESS, and POSITION.

COMPANY_STATS table with these columns: NBEMP, NBPRODUCT, and REVENUE.

The first trigger increments the number of employees each time a new person is hired; that is, each time a new row is inserted into the EMPLOYEE table:

```
CREATE TRIGGER NEW_HIRED
   AFTER INSERT ON EMPLOYEE
   FOR EACH ROW MODE DB2SQL
   UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

The second trigger decrements the number of employees each time an employee leaves the company; that is, each time a row is deleted from the table EMPLOYEE:

```
CREATE TRIGGER FORMER_EMP
   AFTER DELETE ON EMPLOYEE
   FOR EACH ROW MODE DB2SQL
   UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1
```

Example 2: Create a trigger that ensures that whenever a parts record is updated, the following check and (if necessary) action is taken:

If the on-hand quantity is less than 10% of the maximum stocked quantity, then issue a shipping request ordering the number of items for the affected part to be equal to the maximum stocked quantity minus the on-hand quantity.

The trigger will interact with the PARTS table with these columns: PARTNO, DESCRIPTION, ON_HAND, MAX_STOCKED, and PRICE.

ISSUE_SHIP_REQUEST is a user-defined function that sends an order form for additional parts to the appropriate company.

```
CREATE TRIGGER REORDER
     AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
     REFERENCING NEW AS N
     FOR EACH ROW MODE DB2SQL
     WHEN (N.ON_HAND < 0.10 * N.MAX_STOCKED)
     BEGIN ATOMIC
     VALUES(ISSUE_SHIP_REQUEST(N.MAX_STOCKED - N.ON_HAND, N.PARTNO));
     END
```

Example 3: Create a trigger that will cause an error when an update occurs that would result in a salary increase greater than ten percent of the current salary.

```
   CREATE TRIGGER RAISE_LIMIT
     AFTER UPDATE OF SALARY ON EMPLOYEE
     REFERENCING NEW AS N OLD AS O
     FOR EACH ROW MODE DB2SQL
     WHEN (N.SALARY > 1.1 * O.SALARY)
            SIGNAL SQLSTATE '75000' ('Salary increase>10%')
```

Example 4: Consider an application which records and tracks changes to stock prices. The database contains two tables, CURRENTQUOTE and QUOTEHISTORY.

```
Tables: CURRENTQUOTE (SYMBOL, QUOTE, STATUS)
        QUOTEHISTORY (SYMBOL, QUOTE, QUOTE_TIMESTAMP)
```

When the QUOTE column of CURRENTQUOTE is updated, the new quote should be copied, with a timestamp, to the QUOTEHISTORY table. Also, the STATUS column of CURRENTQUOTE should be updated to reflect whether the stock is:
1. rising in value;
2. at a new high for the year;
3. dropping in value;
4. at a new low for the year;
5. steady in value.

CREATE TRIGGER statements that accomplish this are as follows.
- Trigger Definition to set the status:

```
      CREATE TRIGGER STOCK_STATUS
        NO CASCADE BEFORE UPDATE OF QUOTE ON CURRENTQUOTE
        REFERENCING NEW AS NEWQUOTE OLD AS OLDQUOTE
        FOR EACH ROW MODE DB2SQL
        BEGIN ATOMIC
           SET NEWQUOTE.STATUS =
        CASE
           WHEN NEWQUOTE.QUOTE >
                 (SELECT MAX(QUOTE) FROM QUOTEHISTORY
                  WHERE SYMBOL = NEWQUOTE.SYMBOL
                  AND YEAR(QUOTE_TIMESTAMP) = YEAR(CURRENT DATE) )
              THEN 'High'
           WHEN NEWQUOTE.QUOTE <
                 (SELECT MIN(QUOTE) FROM QUOTEHISTORY
                  WHERE SYMBOL = NEWQUOTE.SYMBOL
                  AND YEAR(QUOTE_TIMESTAMP) = YEAR(CURRENT DATE) )
              THEN 'Low'
           WHEN NEWQUOTE.QUOTE > OLDQUOTE.QUOTE
              THEN 'Rising'
           WHEN NEWQUOTE.QUOTE < OLDQUOTE.QUOTE
              THEN 'Dropping'
           WHEN NEWQUOTE.QUOTE = OLDQUOTE.QUOTE
              THEN 'Steady'
        END;
  END
```

- Trigger Definition to record change in QUOTEHISTORY table:

```
CREATE TRIGGER RECORD_HISTORY
  AFTER UPDATE OF QUOTE ON CURRENTQUOTE
  REFERENCING NEW AS NEWQUOTE
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
     INSERT INTO QUOTEHISTORY
        VALUES (NEWQUOTE.SYMBOL, NEWQUOTE.QUOTE, CURRENT TIMESTAMP);
  END
```

CREATE TYPE (Structured)

The CREATE TYPE statement defines a user-defined structured type. A user-defined structured type may include zero or more attributes. A structured type may be a subtype allowing attributes to be inherited from a supertype. Successful execution of the statement generates methods, for retrieving and updating values of attributes. Successful execution of the statement also generates functions, for constructing instances of a structured type used in a column, for casting between the reference type and its representation type, and for supporting the comparison operators (=, <>, <, <=, >, and >=) on the reference type.

The CREATE TYPE statement also defines any method specifications for user-defined methods to be used with the user-defined structured type.

Invocation

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

Authorization

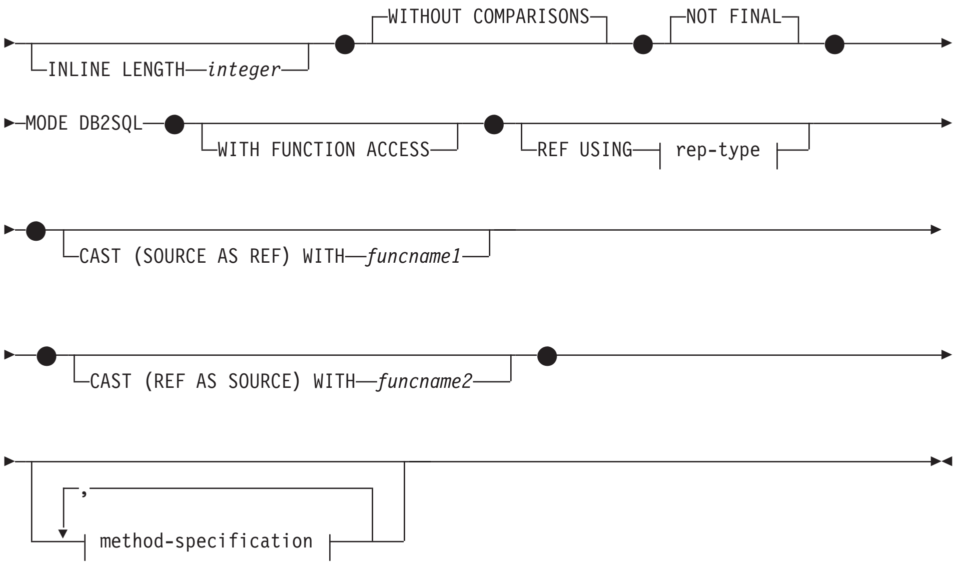
The privileges held by the authorization ID of the statement must include as least one of the following:

- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the database, if the schema name of the type does not refer to an existing schema.
- CREATEIN privilege on the schema, if the schema name of the type refers to an existing schema.

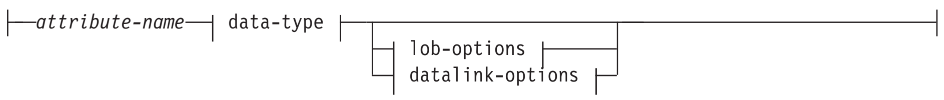
If UNDER is specified and the authorization ID of the statement is not the same as the definer of the root type of the type hierarchy, then SYSADM or DBADM authority is required.

Syntax

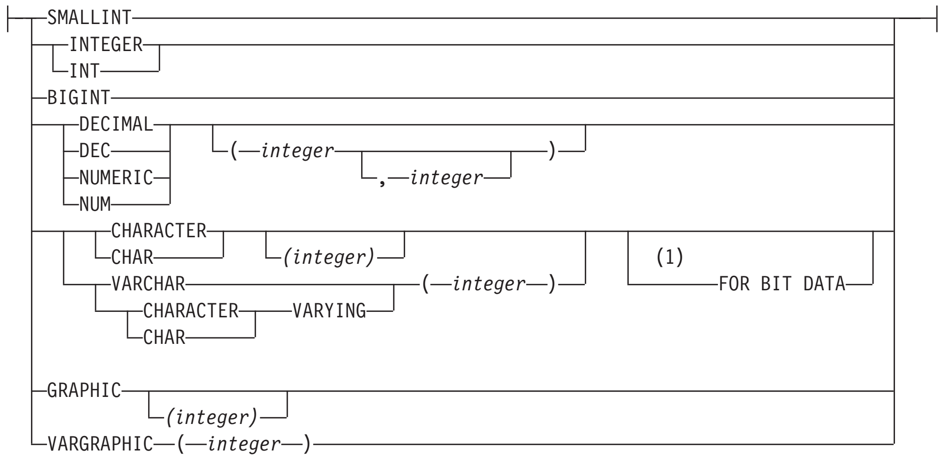
```
>>──CREATE TYPE──type-name──┬──────────────────────────┬──────────────────>
                            └─UNDER──supertype-name────┘

                                                   ┌─INSTANTIABLE─────┐
>──┬─────────────────────────────────────────────┬─●─┼──────────────────┼─●────────>
   │       ┌─,──────────────────────────┐        │   └─NOT INSTANTIABLE─┘
   │       ▼                            │        │
   └─AS──(───┤ attribute-definition ├───┴──)─────┘
```



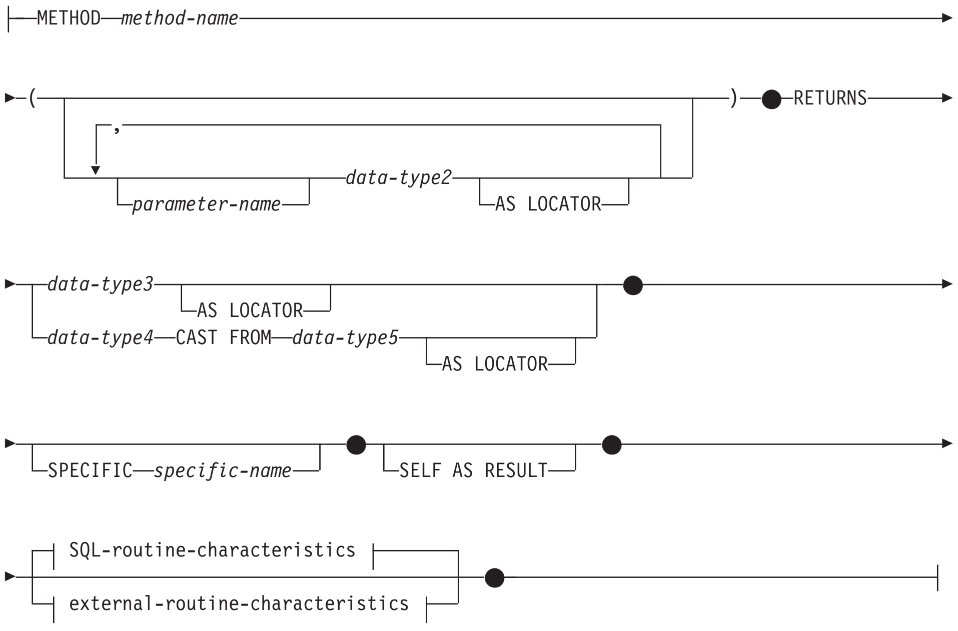
attribute-definition:



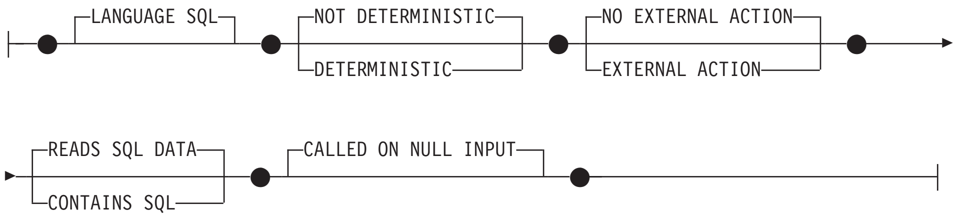
rep-type:



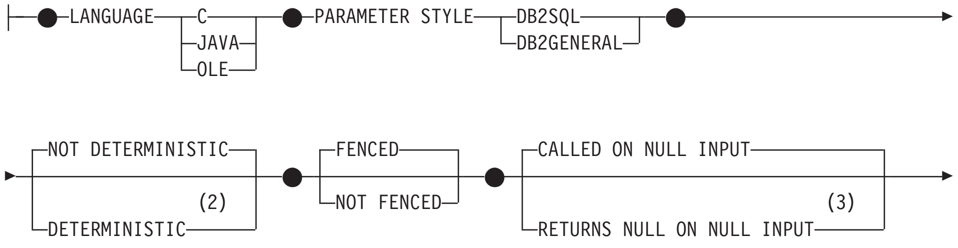
method-specification:

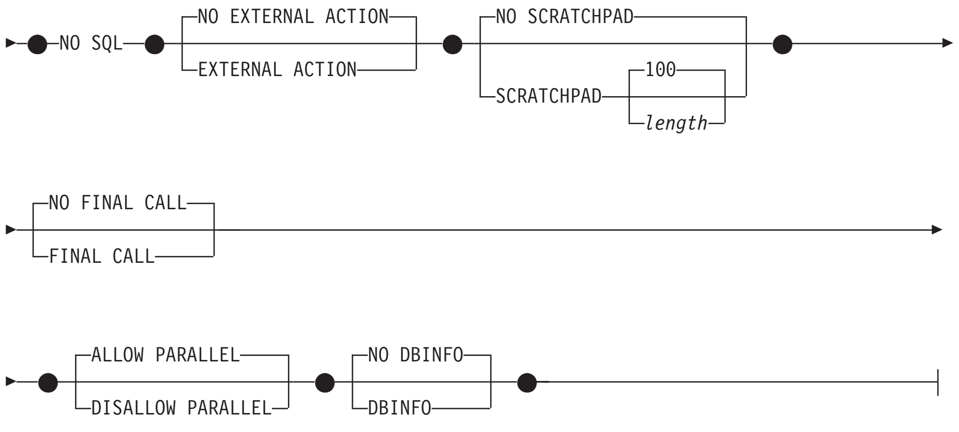


SQL-routine-characteristics:



external-routine-characteristics:





Notes:

1 The FOR BIT DATA clause may be specified in random order with the other column constraints that follow.

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

type-name
Names the type. The name, including the implicit or explicit qualifier, must not identify any other type (built-in, structured, or distinct) already described in the catalog. The unqualified name must not be the same as the name of a built-in data type or BOOLEAN (SQLSTATE 42918). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

The schema name (implicit or explicit) must not be greater than 8 bytes (SQLSTATE 42622).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *type-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in "Basic Predicate" on page 187.

If a two-part *type-name* is specified, the schema name cannot begin with ″SYS″; otherwise, an error (SQLSTATE 42939) is raised.

UNDER *supertype-name*
Specifies that this structured type is a subtype under the specified *supertype-name*. The *supertype-name* must identify an existing structured type (SQLSTATE 42704). If *supertype-name* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The structured type includes all the attributes of the supertype followed by the additional attributes given in the *attribute-definition*.

attribute-definition
Defines the attributes of the structured type.

attribute-name
The name of an attribute. The *attribute-name* cannot be the same as any other attribute of this structured type or any supertype of this structured type (SQLSTATE 42711).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as an *attribute-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in “Basic Predicate” on page 187.

data-type
The data type of the attribute. It is one of the data types listed under “CREATE TABLE” on page 712 other than LONG VARCHAR, LONG VARGRAPHIC, or a distinct type based on LONG VARCHAR or LONG VARGRAPHIC (SQLSTATE 42601). The data type must identify an existing data type (SQLSTATE 42704). If *data-type* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The description of various data types is given in “CREATE TABLE” on page 712. If the attribute data type is a reference type, the target type of the reference must be a structured type that exists, or is created by this statement (SQLSTATE 42704).

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

To prevent type definitions that would, at runtime, permit an instance of the type to directly or indirectly contain another instance of the same type or one of its subtypes, a type can not be defined such that one of its attribute types directly or indirectly uses itself (SQLSTATE 428EP). See “Structured Types” on page 88 for more information.

lob-options
Specifies the options associated with LOB types (or distinct types based on LOB types). For a detailed description of *lob-options*, see "CREATE TABLE" on page 712.

datalink-options
Specifies the options associated with DATALINK types (or distinct types based on DATALINK types). For a detailed description of *datalink-options*, see "CREATE TABLE" on page 712.

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

INSTANTIABLE or NOT INSTANTIABLE
Determines whether an instance of the structured type can be created. Implications of not instantiable structured types are:

- no constructor function is generated for a non-instantiable type
- a non-instantiable type cannot be used as the type of a table or view (SQLSTATE 428DP)
- a non-instantiable type can be used as the type of a column (only null values or instances of instantiable subtypes can be inserted into the column.

To create instances of a non-instantiable type, instantiable subtypes must be created. If NOT INSTANTIABLE is specified, no instance of the new type can be created.

INLINE LENGTH *integer*
This option indicates the maximum size (in bytes) of a structured type column instance to store inline with the rest of the values in the row of a table. Instances of a structured type or its subtypes, that are larger than the specified inline length, are stored separately from the base table row, similar to the way that LOB values are handled.

If the specified INLINE LENGTH is smaller than the size of the result of the constructor function for the newly-created type (32 bytes plus 10 bytes per attribute) and smaller than 292 bytes, an error results (SQLSTATE 429B2). Note that the number of attributes includes all attributes inherited from the supertype of the type.

The INLINE LENGTH for the type, whether specified or a default value, is the default inline length for columns that use the structured type. This default can be overridden at CREATE TABLE time.

INLINE LENGTH has no meaning when the structured type is used as the type of a typed table.

The default INLINE LENGTH for a structured type is calculated by the system. In the formula given below, the following terms are used:

short attribute
: refers to an attribute with any of the following data types: SMALLINT, INTEGER, BIGINT, REAL, DOUBLE, FLOAT, DATE, or TIME. Also included are distinct types or reference types based on these types.

non-short attribute
: refers to an attribute of any of the remaining data types, or distinct types based on those data types.

The system calculates the default inline length as follows:

1. Determine the added space requirements for non-short attributes using the following formula:

   space_for_non_short_attributes = SUM(*attributelength* + n)

   n is defined as:

   - 0 bytes for nested structured type attributes
   - 2 bytes for non-LOB attributes
   - 9 bytes for LOB attributes

   attributelength is based on the data type specified for the attribute as shown in Table 25.
2. Calculate the total default inline length using the following formula:

   default_length(*structured_type*) = (*number_of_attributes* * 10) + 32 + *space_for_non-short_attributes*

   number_of_attributes is the total number of attributes for the structured type, including attributes that are inherited from its supertype. However, *number_of_attributes* does not include any attributes defined for any subtype of *structured_type*.

Table 25. Byte Counts for Attribute Data Types

Attribute Data Type	Byte Count
DECIMAL	The integral part of (p/2)+1, where p is the precision
CHAR(n)	n
VARCHAR(n)	n
GRAPHIC(n)	n * 2
VARGRAPHIC(n)	n * 2
TIMESTAMP	10
DATALINK(n)	n + 54

Table 25. Byte Counts for Attribute Data Types (continued)

LOB Type	Each LOB attribute has a LOB descriptor in the structured type instance that points to the location of the actual value. The size of the descriptor varies according to the maximum length defined for the LOB attribute	
	Maximum LOB Length	LOB Descriptor Size
	1 024	72
	8 192	96
	65 536	120
	524 000	144
	4 190 000	168
	134 000 000	200
	536 000 000	224
	1 070 000 000	256
	1 470 000 000	280
	2 147 483 647	316
Distinct Type	Length of the source type of the distinct type	
Reference Type	Length of the built-in data type on which the reference type is based.	
Structured Type	inline_length(*attribute_type*)	

WITHOUT COMPARISONS
Indicates that there are no comparison functions supported for instances of the structured type.

NOT FINAL
Indicates that the structured type may be used as a supertype.

MODE DB2SQL
This clause is required and allows for direct invocation of the constructor function on this type.

WITH FUNCTION ACCESS
Indicates that all methods of this type and its subtypes, including methods created in the future, can be accessed using functional notation. This clause can be specified only for the root type of a structured type hierarchy (the UNDER clause is not specified) (SQLSTATE 42613). This clause is provided to allow the use of functional notation for those applications that prefer this form of notation over method invocation notation.

REF USING *rep-type*
Defines the built-in data type used as the representation (underlying data type) for the reference type of this structured type and all its subtypes. This clause can only be specified for the root type of a structured type

hierarchy (UNDER clause is not specified) (SQLSTATE 42613). The *rep-type* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, DATALINK, or structured type, and must have a length less than or equal to 255 bytes (SQLSTATE 42613).

If this clause is not specified for the root type of a structured type hierarchy, then REF USING VARCHAR(16) FOR BIT DATA is assumed.

CAST (SOURCE AS REF) WITH *funcname1*
Defines the name of the system-generated function that casts a value with the data type *rep-type* to the reference type of this structured type. A schema name must not be specified as part of *funcname1* (SQLSTATE 42601). The cast function is created in the same schema as the structured type. If the clause is not specified, the default value for *funcname1* is *type-name* (the name of the structured type). A function signature matching *funcname1(rep-type)* must not already exist in the same schema (SQLSTATE 42710).

CAST (REF AS SOURCE) WITH *funcname2*
Defines the name of the system-generated function that casts a reference type value for this structured type to the data type *rep-type*. A schema name must not be specified as part of *funcname2* (SQLSTATE 42601). The cast function is created in the same schema as the structured type. If the clause is not specified, the default value for *funcname2* is *rep-type* (the name of the representation type).

method-specification
Defines the methods for this type. A method cannot actually be used until it is given a body with a CREATE METHOD statement (SQLSTATE 42884).

method-name
Names the method being defined. It must be an unqualified SQL identifier (SQLSTATE 42601). The method name is implicitly qualified with the schema used for CREATE TYPE.

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *method-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in "Basic Predicate" on page 187.

In general, the same name can be used for more than one method if there is some difference in their signatures.

parameter-name
Identifies the parameter name. It cannot be SELF, which is the name for the implicit subject parameter of a method (SQLSTATE 42734). If the method is an SQL method, all its parameters must have names (SQLSTATE 42629).

data-type2

Specifies the data type of each parameter. One entry in the list must be specified for each parameter that the method will expect to receive. No more than 90 parameters are allowed, including the implicit SELF parameter. If this limit is exceeded, an error is raised (SQLSTATE 54023).

SQL data type specifications and abbreviations which may be specified as a column-type in a CREATE TABLE statement and have a correspondence in the language that is being used to write the method may be specified. Refer to the language-specific sections of the Application Development Guide for details on the mapping between SQL data types and host language data types with respect to user-defined functions and methods.

Note: If the SQL data type in question is a structured type, there is no default mapping to a host language data type. A user-defined transform function must be used to create a mapping between the structured type and the host language data type.

DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815). For alternatives to using DECIMAL, refer to the *Application Development Guide*.

REF may be specified, but it does not have a defined scope. Inside the body of the method, a reference-type can be used in a path-expression only by first casting it to have a scope. Similarly, a reference returned by a method can be used in a path-expression only by first casting it to have a scope.

AS LOCATOR

For LOB types or distinct types which are based on a LOB type, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed to the method instead of the actual value. This saves greatly in the number of bytes passed to the method, and may save as well in performance, particularly in the case where only a few bytes of the value are actually of interest to the method. Use of LOB locators is described in the *Application Development Guide*.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

RETURNS
This mandatory clause identifies the method's result.

data-type3
Specifies the data type of the method's result. In this case, exactly the same considerations apply as for the parameters of methods described above under data-type2.

AS LOCATOR
For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the method instead of the actual value.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

data-type4 **CAST FROM** *data-type5*
Specifies the data type of the method's result.

This clause is used to return a different data type to the invoking statement from the data type returned by the method code. The *data-type5* must be castable to the *data-type4* parameter. If it is not castable, an error is raised (SQLSTATE 42880).

Since the length, precision or scale for *data-type4* can be inferred from *data-type5*, it not necessary (but still permitted) to specify the length, precision, or scale for parameterized types specified for *data-type4*. Instead, empty parentheses may be used (VARCHAR(), for example). FLOAT() cannot be used (SQLSTATE 42601), since the parameter value indicates different data types (REAL or DOUBLE).

A distinct type is not valid as the type specified in data-type5 (SQLSTATE 42815).

The cast operation is also subject to runtime checks that might result in conversion errors being raised.

AS LOCATOR
For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the method instead of the actual value.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

SPECIFIC *specific-name*
Provides a unique name for the instance of the method that is being defined. This specific name can be used when creating the method body or dropping the method. It can never be used to invoke the method. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a schema-name followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another specific method name that exists at the application server; otherwise an error is raised (SQLSTATE 42710).

The *specific-name* may be the same as an existing *method-name*.

If no qualifier is specified, the qualifier that was used for *type-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *type-name* or an error is raised (SQLSTATE 42882).

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

SELF AS RESULT
Identifies this method as a type-preserving method, which means the following:

- The declared return type must be the same as the declared subject-type (SQLSTATE 428EQ).
- When an SQL statement is compiled and resolves to a type preserving method, the static type of the result of the method is the same as the static type of the subject argument.
- The method must be implemented in such a way that the dynamic type of the result is the same as the dynamic type of the subject argument (SQLSTATE 2200G) and the result may also not be NULL (SQLSTATE 22004).

SQL-routine-characteristics
Specifies the characteristics of the method body that will be defined for this type using CREATE METHOD.

LANGUAGE SQL
This clause is used to indicate that the method is written in SQL with a single RETURN statement. The method body is specified using the CREATE METHOD statement.

NOT DETERMINISTIC or DETERMINISTIC
This optional clause specifies whether the method always returns the same results for given argument values (DETERMINISTIC) or whether the method depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC method must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical

inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. NOT DETERMINISTIC must be explicitly or implicitly specified if the body of the method accesses a special register, or calls another non-deterministic routine (SQLSTATE 428C2).

NO EXTERNAL ACTION or EXTERNAL ACTION

This optional clause specifies whether or not the method takes some action that changes the state of an object not managed by the database manager. Optimizations that assume methods have no external impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

READS SQL DATA or CONTAINS SQL

Indicates what type of SQL statements can be executed. Because the SQL statement supported is the RETURN statement, the distinction has to do with whether or not the expression is a subquery.

READS SQL DATA

Indicates that SQL statements that do not modify SQL data can be executed by the method (SQLSTATE 42985). Nicknames cannot be referenced in the SQL statement (SQLSTATE 42997).

CONTAINS SQL

Indicates that SQL statements that neither read nor modify SQL data can be executed by the method (SQLSTATE 42985).

CALLED ON NULL INPUT

This optional clause indicates that regardless of whether any arguments are null, the user-defined method is called. It can return a null value or a normal (non-null) value. However, responsibility for testing for null argument values lies with the method.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for family compatibility.

external-routine-characteristics

LANGUAGE

This mandatory clause is used to specify the language interface convention to which the user-defined method body is written.

C This means the database manager will call the user-defined method as if it were a C function. The user-defined method must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA

This means the database manager will call the user-defined method as a method in a Java class.

OLE
This means the database manager will call the user-defined method as if it were a method exposed by an OLE automation object. The method must conform with the OLE automation data types and invocation mechanism as described in the OLE Automation Programmer's Reference.

LANGUAGE OLE is only supported for user-defined methods stored in Windows 32-bit operating systems.

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

DB2SQL
Used to specify the conventions for passing parameters to and returning the value from external methods that conform to C language calling and linkage conventions or methods exposed by OLE automation objects. This must be specified when either LANGUAGE C or LANGUAGE OLE is used.

DB2GENERAL
Used to specify the conventions for passing parameters to and returning the value from external methods that are defined as a method in a Java class. This can only be specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

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

DETERMINISTIC or NOT DETERMINISTIC
This optional clause specifies whether the method always returns the same results for given argument values (DETERMINISTIC) or whether the method depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC method must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC.

An example of a NOT DETERMINISTIC method would be a method that randomly returns a serial number of an employee in a department. An example of a DETERMINISTIC method would be a method that calculates the area of a polygon.

FENCED or NOT FENCED
This clause specifies whether the method is considered ″safe″ to run in

the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a method is registered as FENCED, the database manager insulates its internal resources (data buffers, for example) from access by the method. Most methods will have the option of running as FENCED or NOT FENCED. In general, a method running as FENCED will not perform as well as a similar one running as NOT FENCED.

Note: Use of NOT FENCED for methods not adequately checked out can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user defined methods are used.

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

Most methods should be able to run either as FENCED or NOT FENCED. Only FENCED can be specified for a method with LANGUAGE OLE (SQLSTATE 42613).

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

To change from FENCED to NOT FENCED, the method must be re-registered, by first dropping it and then recreating it.

Either SYSADM authority, DBADM authority or a special authority (CREATE_NOT_FENCED) is required to register a method 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 method if any of the non-subject arguments is null.

If RETURNS NULL ON NULL INPUT is specified, and if at execution time any one of the method's arguments is null, the method is not called and the result is the null value.

If CALLED ON NULL INPUT is specified, then regardless of the number of null arguments, the method is called. It can return a null value or a normal (non-null) value. However, responsibility for testing for null argument values lies with the method.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

There are two cases in which this specification is ignored:

- If the subject argument is null, in which case the method is not executed and the result is null
- If the method is defined to have no parameters, in which case this null argument condition cannot occur.

NO SQL

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

EXTERNAL ACTION or NO EXTERNAL ACTION

This optional clause specifies whether or not the method takes some action that changes the state of an object not managed by the database manager. Optimizations that assume methods have no external impacts are prevented by specifying EXTERNAL ACTION.

NO SCRATCHPAD or SCRATCHPAD *length*

This optional clause may be used to specify whether a scratchpad is to be provided for an external method. It is strongly recommended that methods be re-entrant, so a scratchpad provides a means for the method to ″save state″ from one call to the next.

If SCRATCHPAD is specified, then at the first invocation of the user-defined method, memory is allocated for a scratchpad to be used by the external method. This scratchpad has the following characteristics:

- *length*, if specified, sets the size in bytes of the scratchpad and must be between 1 and 32 767 (SQLSTATE 42820). The default value is 100.
- It is initialized to all X’00’’s.
- Its scope is the SQL statement. There is one scratchpad per reference to the external method in the SQL statement.

So, if method X in the following statement is defined with the SCRATCHPAD keyword, three scratchpads would be assigned.

```
SELECT A, X..(A) FROM TABLEB
  WHERE X..(A) > 103 OR X..(A) < 19
```

If ALLOW PARALLEL is specified or defaulted to, then the scope is different from the above. If the method is executed in multiple partitions, a scratchpad would be assigned in each partition where the method is processed, for each reference to the method in the SQL statement. Similarly, if the query is executed with intra-partition parallelism enabled, more than three scratchpads may be assigned.

The scratchpad is persistent. Its content is preserved from one external method call to the next. Any changes made to the scratchpad by the external method on one call will be present on the next call. The database manager initializes scratchpads at the beginning of execution of each SQL statement. The database manager may reset scratchpads at the beginning of execution of each subquery. The system issues a final call before resetting a scratchpad if the FINAL CALL option is specified.

The scratchpad can be used as a central point for system resources (memory, for example) which the external method might acquire. The method could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

In such a case where system resource is acquired, the FINAL CALL keyword should also be specified; this causes a special call to be made at end-of-statement to allow the external method to free any system resources acquired.

If SCRATCHPAD is specified, then on each invocation of the user-defined method, an additional argument is passed to the external method which addresses the scratchpad.

If NO SCRATCHPAD is specified, then no scratchpad is allocated or passed to the external method.

NO FINAL CALL or FINAL CALL

This optional clause specifies whether a final call is to be made to an external method. The purpose of such a final call is to enable the external method to free any system resources it has acquired. It can be useful in conjunction with the SCRATCHPAD keyword in situations where the external method acquires system resources such as memory and anchors them in the scratchpad.

If FINAL CALL is specified, then at execution time, an additional argument is passed to the external method which specifies the type of call. The types of calls are:

- Normal call: SQL arguments are passed and a result is expected to be returned.
- First call: the first call to the external method for this specific reference to the method in this specific SQL statement. The first call is a normal call.

- Final call: a final call to the external method to enable the method to free up resources. The final call is not a normal call. This final call occurs at the following times:
  - End-of-statement: this case occurs when the cursor is closed for cursor-oriented statements, or when the statement is through executing otherwise.
  - End-of-transaction: This case occurs when the normal end-of-statement does not occur. For example, the logic of an application may for some reason bypass the close of the cursor.

  If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made at the subsequent close of the cursor or at the end of the application.

If NO FINAL CALL is specified, then no ″call type″ argument is passed to the external method, and no final call is made.

ALLOW PARALLEL or DISALLOW PARALLEL

This optional clause specifies whether, for a single reference to the method, the invocation of the method can be parallelized. In general, the invocations of most scalar methods should be parallelizable, but there may be methods (such as those depending on a single copy of a scratchpad) that cannot. If either ALLOW PARALLEL or DISALLOW PARALLEL are specified for a method, then DB2 will accept this specification.

The following questions should be considered in determining which keyword is appropriate for the method:.

- Are all the method invocations completely independent of each other? If YES, then specify ALLOW PARALLEL.
- Does each method invocation update the scratchpad, providing value(s) that are of interest to the next invocation (the incrementing of a counter, for example)? If YES, then specify DISALLOW PARALLEL or accept the default.
- Is there some external action performed by the method which should happen only on one partition? If YES, then specify DISALLOW PARALLEL or accept the default.
- Is the scratchpad used, but only so that some expensive initialization processing can be performed a minimal number of times? If YES, then specify ALLOW PARALLEL.

In any case, the body of every external method should be in a directory that is available on every partition of the database.

The syntax diagram indicates that the default value is ALLOW PARALLEL. However, the default is DISALLOW PARALLEL if one or more of the following options is specified in the statement:

- NOT DETERMINISTIC
- EXTERNAL ACTION
- SCRATCHPAD
- FINAL CALL

NO DBINFO or DBINFO

This optional clause specifies whether certain specific information known by DB2 will be passed to the method as an additional invocation-time argument (DBINFO), or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613).

If DBINFO is specified, then a structure is passed to the method 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 runtime authorization ID, regardless of the nested methods in between this method and the application.
- Code page - identifies the database code page.
- Schema name - under the exact same conditions as for Table name, contains the name of the schema; otherwise blank.
- Table name - if and only if the method reference is either the right-hand side of a SET clause in an UPDATE statement, or an item in the VALUES list of an INSERT statement, contains the unqualified name of the table being updated or inserted; otherwise blank.
- Column name - under the exact same conditions as for Table name, contains the name of the column being updated or inserted; otherwise blank.
- Database version/release - identifies the version, release and modification level of the database server invoking the method.
- Platform - contains the server's platform type.
- Table method result column numbers - not applicable to methods.

Refer to *Application Development Guide* for detailed information on the structure and how it is passed to the method.

Notes

- Creating a structured type with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

- A structured subtype defined with no attributes defines a subtype that inherits all its attributes from the supertype. If neither an UNDER clause nor any other attribute is specified, then the type is a root type of a type hierarchy without any attributes.
- The addition of a new subtype to a type hierarchy may cause packages to be invalidated. A package may be invalidated if it depends on a supertype of the new type. Such a dependency is the result of the use of a TYPE predicate or a TREAT specification.
- A structured type may have no more than 4082 attributes (SQLSTATE 54050).
- A method specification is not allowed to have the same signature as a function (comparing the first parameter-type of the function with the subject-type of the method).
- A method MT, with subject-type T, is defined to override another method MS, with subject-type S, if all of the following are true:
  - MT and MS have the same unqualified name and the same number of parameters
  - T is a proper subtype of S
  - The non-subject parameter-types of MT are the same as the corresponding non-subject parameter-types of MS. Note that, ″same″ applies to the basic type, such as VARCHAR, disregarding length and precision.

  No method may override, or be overridden by, another method (SQLSTATE 42745). Futhermore, a function and a method may not be in an overriding relationship. This means that if the function were a method with its first parameter as subject S, it must not override another method of any supertype of S, and it must not be overridden by another method of any subtype of S.
- Creation of a structured type automatically generates a set of functions and methods for use with the type. All the functions and methods are generated in the same schema as the structured type. If the signature of the generated function or method conflicts with or overrides the signature of an existing function in this schema, the statement fails (SQLSTATE 42710). The generated functions or methods cannot be dropped without dropping the structured type (SQLSTATE 42917). The following functions and methods are generated:
  - Functions
    - Reference Comparisons

      Six comparison functions with names =, <>, <, <=, >, >= are generated for the reference type REF(*type-name*). Each of these functions takes two parameters of type REF(*type-name*) and returns true, false, or unknown. The comparison operators for REF(*type-name*) are defined to have the same behavior as the comparison operators for the underlying data type of REF(*type-name*).[85]

The scope of the reference type is not considered in the comparison.

- Cast functions

  Two cast functions are generated to cast between the generated reference type REF(*type-name*) and the underlying data type of this reference type.

  - The name of the function to cast from the underlying type to the reference type is the implicit or explicit *funcname1*.

    The format of this function is:

    ```
    CREATE FUNCTION funcname1 (rep-type)
       RETURNS REF(type-name) ...
    ```

  - The name of the function to cast from the reference type to the underlying type of the reference type is the implicit or explicit *funcname2*.

    The format of this function is:

    ```
    CREATE FUNCTION funcname2 ( REF(type-name) )
       RETURNS rep-type ...
    ```

  For some rep-types, there are additional cast functions generated with *funcname1* to handle casting from constants.

  - If *rep-type* is SMALLINT, the additional generated cast function has the format:

    ```
    CREATE FUNCTION funcname1 (INTEGER)
       RETURNS REF(type-name)
    ```

  - If *rep-type* is CHAR(n), the additional generated cast function has the format:

    ```
    CREATE FUNCTION funcname1 ( VARCHAR(n))
       RETURNS REF(type-name)
    ```

  - If *rep-type* is GRAPHIC(n), the additional generated cast function has the format:

    ```
    CREATE FUNCTION funcname1 (VARGRAPHIC(n))
       RETURNS REF(type-name)
    ```

  The schema name of the structured type must be included in the SQL path (see "SET PATH" on page 1031 or the FUNCPATH BIND option as described in the *Application Development Guide*) for successful use of these operators and cast functions in SQL statements.

- Constructor function

85. All references in a type hierarchy have the same reference representation type. This enables REF(S) and REF(T) to be compared provided that S and T have a common supertype. Since uniqueness of the OID column is enforced only within a table hierarchy, it is possible that a value of REF(T) in one table hierarchy may be ″equal″ to a value of REF(T) in another table hierarchy, even though they reference different rows.

The constructor function is generated to allow a new instance of the type to be constructed. This new instance will have null for all attributes of the type, including attributes that are inherited from a supertype.

The format of the generated constructor function is:

```
CREATE FUNCTION type-name ( )
   RETURNS type-name
  ...
```

If NOT INSTANTIABLE is specified, no constructor function is generated. If the structured type has attributes of type DATALINK, then the invocation of the constructor function fails (SQLSTATE 428ED).

– Methods
  - Observer methods

    An observer method is defined for each attribute of the structured type. For each attribute, the observer method returns the type of the attribute. If the subject is null, the observer method returns a null value of the attribute type.

    For example, the attributes of an instance of the structured type ADDRESS can be observed using `C1..STREET`, `C1..CITY`, `C1..COUNTRY`, and `C1..CODE`.

    The method signature of the generated observer method is as if the following statement had been executed:

```
CREATE TYPE  type-name
      ...
   METHOD attribute-name()
      RETURNS attribute-type
```

    where *type-name* is the structured type name.
  - Mutator methods

    A type-preserving mutator method is defined for each attribute of the structured type. Use mutator methods to change attributes within an instance of a structured type. For each attribute, the mutator method returns a copy of the subject modified by assigning the argument to the named attribute of the copy.

    For example, an instance of the structured type ADDRESS can be mutated using `C1..CODE('M3C1H7')`. If the subject is null, the mutator method raises an error (SQLSTATE 2202D).

    The method signature of the generated mutator method is as if the following statement had been executed:

```
CREATE TYPE type-name
     ...
   METHOD attribute-name (attribute-type)
      RETURNS type-name
```

If the attribute data type is SMALLINT, REAL, CHAR, or GRAPHIC, an additional mutator method is generated in order to support mutation using constants:

- If *attribute-type* is SMALLINT, the additional mutator supports an argument of type INTEGER.
- If *attribute-type* is REAL, the additional mutator supports an argument of type DOUBLE.
- If *attribute-type* is CHAR, the additional mutator supports an argument of type VARCHAR.
- If *attribute-type* is GRAPHIC, the additional mutator supports an argument of type VARGRAPHIC.

- If the structured type is used as a column type, the length of an instance of the type can be no more than 1 GB in length at runtime (SQLSTATE 54049).

- When creating a new subtype for an existing structured type (for use as a column type), any transform functions already written in support of existing related structured types should be re-examined and updated as necessary. Whether the new type is in the same hierarchy as a given type, or in the hierarchy of a nested type, it is likely that the existing transform function associated with this type will need to be modified to include some or all of the new attributes introduced by the new subtype. Generally speaking, since it is the set of transform functions associated with a given type (or type hierarchy) which enables UDF and Client Application access to the structured type, the transform functions should be written to support ALL of the attributes in a given composite hierarchy (that is, including the transitive closure of all subtypes and their nested structured types).

Examples

Example 1: Create a type for department.

```
CREATE TYPE DEPT AS
   (DEPT NAME     VARCHAR(20),
      MAX_EMPS INT)
      REF USING INT
   MODE DB2SQL
```

Example 2: Create a type hierarchy consisting of a type for employees and a subtype for managers.

```
CREATE TYPE EMP AS
  (NAME      VARCHAR(32),
   SERIALNUM INT,
   DEPT      REF(DEPT),
   SALARY    DECIMAL(10,2))
   MODE DB2SQL

CREATE TYPE MGR UNDER EMP AS
  (BONUS     DECIMAL(10,2))
   MODE DB2SQL
```

Example 3: Create a type hierarchy for addresses. Addresses are intended to be used as types of columns. The inline length is not specified, so DB2 will calculate a default length. Encapsulate within the address type definition an external method that calculates how close this address is to a given input address. Create the method body using the CREATE METHOD statement.

```
CREATE TYPE address_t AS
   (STREET     VARCHAR(30),
    NUMBER     CHAR(15),
    CITY       VARCHAR(30),
    STATE      VARCHAR(10))
    NOT FINAL
    MODE DB2SQL
       METHOD SAMEZIP (addr address_t)
       RETURNS INTEGER
       LANGUAGE SQL
       DETERMINISTIC
       CONTAINS SQL
       NO EXTERNAL ACTION

       METHOD DISTANCE (address_t)
       RETURNS FLOAT
       LANGUAGE C
       DETERMINISTIC
       PARAMETER STYLE DB2SQL
       NO SQL
       NO EXTERNAL ACTION

CREATE TYPE germany_addr_t UNDER address_t AS
   (FAMILY_NAME VARCHAR(30))
    NOT FINAL
    MODE DB2SQL

CREATE TYPE us_addr_t UNDER address_t AS
    (ZIP VARCHAR(10))
     NOT FINAL
     MODE DB2SQL
```

Example 4: Create a type that has nested structured type attributes.

```
CREATE TYPE PROJECT AS
      (PROJ_NAME  VARCHAR(20),
       PROJ_ID    INTEGER,
       PROJ_MGR   MGR,
       PROJ_LEAD  EMP,
       LOCATION   ADDR_T,
       AVAIL_DATE DATE)
       MODE DB2SQL
```