ALTER TYPE (Structured)

The ALTER TYPE statement is used to add or drop attributes or method specifications of a user-defined structured type.

Invocation

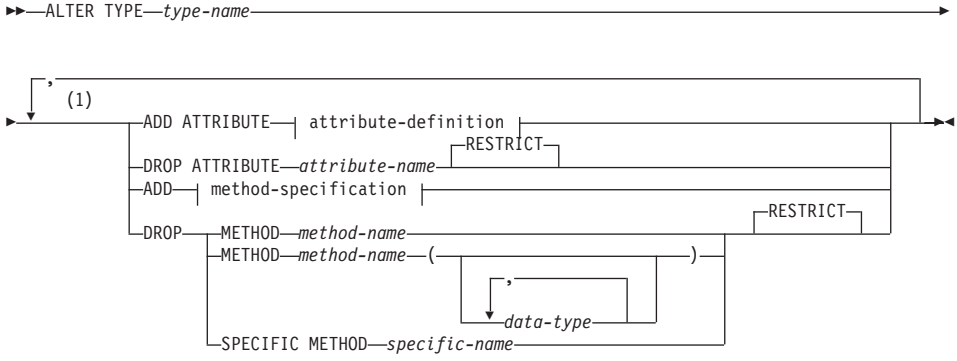
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

Authorization

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

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

Syntax



Notes:

1 If both attributes and methods are added or dropped, all attribute specifications must occur before all method specifications

Description

type-name

Identifies the structured type to be changed. It must be an existing type defined in the catalog (SQLSTATE 42704) and the type must be a structured type (SQLSTATE 428DP). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an

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

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

attribute-definition
For a detailed description of *attribute-definition*, please see "CREATE TYPE (Structured)" on page 792.

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

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

data-type 1
Specifies the data type of the attribute. It is one of the data types listed under CREATE TABLE, other than LONG VARCHAR, LONG VARGRAPHIC, or a distinct type based on LONG VARCHAR or LONG VARGRAPHIC (SQLSTATE 42601). The data type must identify an existing data type (SQLSTATE 42704). If *data-type* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The description of various data types is given in "CREATE TABLE" 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 (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 type view (SQLSTATE 01641).

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

DROP ATTRIBUTE
Drops an attribute of the existing structured type.

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

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

ADD method-specification
Adds a method specification to the type identified by the type-name. The method cannot be used until a separate CREATE METHOD statement is used to give the method a body. For more information about method-specification, see "CREATE TYPE (Structured)" on page 792.

DROP METHOD
Identifies an instance of a method that is to be dropped. The specified method must not have an existing method body (SQLSTATE 428ER). Use the DROP METHOD statement to drop the method body before using ALTER TYPE DROP METHOD.

The specified method must be a method that is described in the catalog (SQLSTATE 42704). Methods implicitly generated by the CREATE TYPE statement (such as mutators and observers) cannot be dropped (SQLSTATE 42917).

There are several ways available to identify the method specification to be dropped:

METHOD *method-name*
Identifies the particular method, and is valid only if there is exactly one method instance with name *method-name* and subject type *type-name*. The method thus identified may have any number of parameters. If no method by this name exists for the type *type-name*, an error is raised (SQLSTATE 42704). If there is more than one method with the name *method-name* for the named data type, an error is raised (SQLSTATE 42854).

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

method-name
The name of the method to be dropped for the specific type. The name must be an unqualified identifier.

(data-type,...)
Must match the data types that were specified in the corresponding positions of the method-specification when the method was defined. The number of data types and the logical concatenation of the data types is used to identify the specific method instance which is to be dropped.

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

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

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

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

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

SPECIFIC METHOD *specific-name*
Identifies the particular method that is to be dropped, using the specific name either given or defaulted to when the method was defined. If *specific-name* is an unqualified name, the method is implicitly qualified with the schema of the data type specified for *type-name*. The specific-name must identify a method for the type *type-name*; otherwise an error is raised (SQLSTATE 42704).

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

Rules

- Adding or dropping an attribute is not allowed for type *type-name* (SQLSTATE 55043) if either:

  - the type or one of its subtypes is the type of an existing table or view or
  - there exists a column of a table whose type directly or indirectly uses *type-name*. The terms *directly uses* and *indirectly uses* are defined in "Structured Types" on page 88
  - the type or one of its subtypes is used in an index extension.
- A type may not be altered by adding attributes so that the total number of attributes for the type, or any of its subtypes, exceeds 4082 (SQLSTATE 54050).
- ADD ATTRIBUTE option:
  - ADD ATTRIBUTE generates observer and mutator methods for the new attribute. These methods are similar to those generated when a structured type is created, as described in "CREATE TYPE (Structured)" on page 792. If these methods conflict with or override any existing methods or functions, the ALTER TYPE statement fails (SQLSTATE 42745).
  - If the INLINE LENGTH for the type (or any of its subtypes) was explicitly specified by the user with a value less than 292, and the attributes added cause the specified inline length to be less than the size of the result of the constructor function for the altered type (32 bytes plus 10 bytes per attribute), then an error results (SQLSTATE 42611).
- DROP ATTRIBUTE option:
  - An attribute that is inherited from an existing supertype cannot be dropped (SQLSTATE 428DJ).
  - DROP ATTRIBUTE drops the mutator and observer methods of the dropped attributes, and checks dependencies on those dropped methods.

Notes

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

or one of its subtypes in a column definition, should be large enough to account for the possible increase in length of the instantiated object.

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

Examples

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

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

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

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

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

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

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

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

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

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

```
ALTER TYPE EMP ...
   ADD METHOD BONUS (RATE DOUBLE)
      RETURNS INTEGER
```

```
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
```

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

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

See "CREATE METHOD" on page 676 for a description of this statement.

ALTER USER MAPPING

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

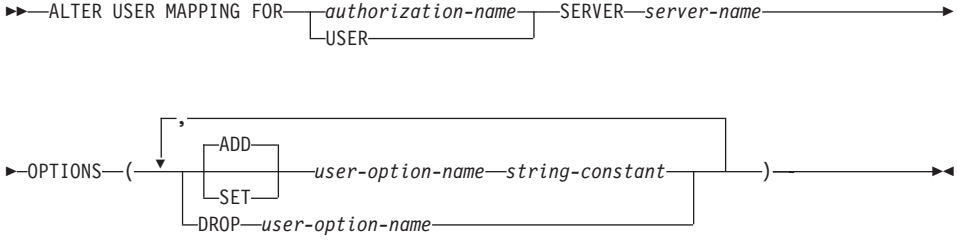
Invocation

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

Authorization

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

Syntax



Description

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

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

SERVER *server-name*
Identifies the data source accessible under the remote authorization ID that maps to the local authorization ID that's denoted by *authorization-name* or referenced by USER.

OPTIONS
Indicates what user options are to be enabled, reset, or dropped for the

mapping that is being altered. Refer to "User Options" on page 1254 for descriptions of *user-option-name*s and their settings.

ADD
Enables a user option.

SET
Changes the setting of a user option.

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

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

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

Notes

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

Examples

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

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

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

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

ALTER VIEW

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

Invocation

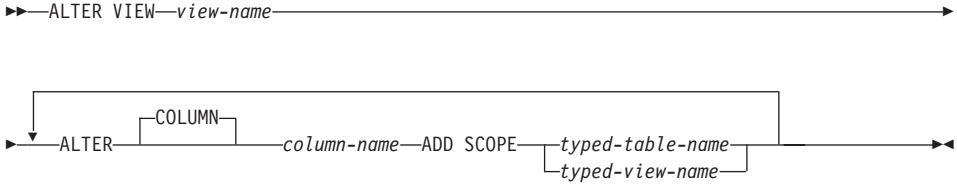
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

Authorization

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

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

Syntax



Description

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

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

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

typed-table-name
The name of a typed table. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM).

No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

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

BEGIN DECLARE SECTION

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

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in REXX.

Authorization

None required.

Syntax

Description

The BEGIN DECLARE SECTION statement may be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. It is used to indicate the beginning of a host variable declaration section. A host variable section ends with an END DECLARE SECTION statement (see "END DECLARE SECTION" on page 894).

Rules

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

Examples

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

```
EXEC SQ  BEGIN DECLARE SECTION;
  short                         hv_smint;
  struct {
```

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

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

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

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

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

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

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

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

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

CALL

Invokes a procedure stored at the location of a database. A stored procedure, for example, executes at the location of the database, and returns data to the client application.

Programs using the SQL CALL statement are designed to run in two parts, one on the client and the other on the server. The server procedure at the database runs within the same transaction as the client application. If the client application and stored procedure are on the same partition, the stored procedure is executed locally.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. However, the procedure name may be specified via a host variable and this, coupled with the use of the USING DESCRIPTOR clause, allows both the procedure name and the parameter list to be provided at run time; thus achieving the same effect as a dynamically prepared statement.

Authorization

The authorization rules vary according to the server at which the procedure is stored.

DB2 Universal Database:

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

- EXECUTE privilege for the package associated with the stored procedure
- CONTROL privilege for the package associated with the stored procedure
- SYSADM or DBADM authority

DB2 Universal Database for OS/390:

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

- EXECUTE privilege for the package associated with the stored procedure
- Ownership of the package associated with the stored procedure
- PACKADM authority for the package's collection
- SYSADM authority

DB2 for AS/400:

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

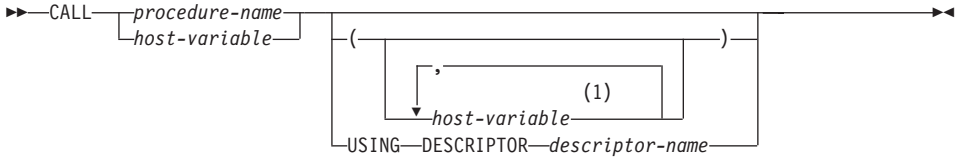
- If the stored procedure is written in REXX:

- The system authorities *OBJOPR and *READ on the source file associated with the procedure
- The system authority *EXECUTE on the library containing the source file and the system authority *USE to the CL command

• If the stored procedure is not written in REXX:
- The system authority *EXECUTE on both the program associated with the procedure and on the library containing that program

• Administrative authority

Syntax



Notes:

1 Stored procedures located at DB2 Universal Database for OS/390 and DB2 Universal Database for AS/400 servers and invoked by DB2 Universal Database for OS/390 or DB2 Universal Database for AS/400 clients support additional sources for procedure arguments (for example constant values). However, if the stored procedure is located on a DB2 Universal Database or the procedure is invoked from a DB2 Universal Database client, all arguments must be provided via host variables.

Description

procedure-name or *host-variable*
Identifies the procedure to call. The procedure name may be specified either directly or within a host variable. The procedure identified must exist at the current server (SQLSTATE 42724).

If *procedure-name* is specified it must be an ordinary identifier not greater than 254 bytes. Since this can only be an ordinary identifier, it cannot contain blanks or special characters and the value is converted to upper case. Thus, if it is necessary to use lower case names, blanks or special characters, the name must be specified via a *host-variable*.

If *host-variable* is specified, it must be a character-string variable with a length attribute that is not greater than 254 bytes, and it must not include an indicator variable. Note that the value is **not** converted to upper case. *procedure-name* must be left-justified.

The procedure name can take one of several forms. The forms supported vary according to the server at which the procedure is stored.

DB2 Universal Database:

procedure-name The name (with no extension) of the procedure to execute. The procedure invoked is determined as follows.

1. The *procedure-name* is used both as the name of the stored procedure library and the function name within that library. For example, if *procedure-name* is `proclib`, the DB2 server will load the stored procedure library named `proclib` and execute the function routine `proclib()` within that library.

   In UNIX-based systems, the DB2 server finds the stored procedure library in the default directory `sqllib/function`. Unfenced stored procedures are in the `sqllib/function/unfenced` directory.

   In OS/2, the location of the stored procedures is specified by the LIBPATH variable in the CONFIG.SYS file. Unfenced stored procedures are in the `sqllib\dll\unfenced` directory.

2. If the library or function could not be found, the *procedure-name* is used to search the defined procedures (in SYSCAT.PROCEDURES) for a matching procedure. A matching procedure is determined using the steps that follow.

   a. Find the procedures from the catalog (SYSCAT.PROCEDURES) where the PROCNAME matches the *procedure-name* specified and the PROCSCHEMA is a schema name in the SQL path (CURRENT PATH special register). If the schema name is explicitly specified, the SQL path is ignored and only procedures with the specified schema name are considered.

   b. Next, eliminate any of these procedures that do not have the same number of parameters as the number of arguments specified in the CALL statement.

   c. Chose the remaining procedure that is earliest in the SQL path.

d. If there are no remaining procedures after step 2, an error is returned (SQLSTATE 42884).

Once the procedure is selected, DB2 will invoke the procedure defined by the external name.

procedure-library!function-name
The exclamation character (!), acts as a delimiter between the library name and the function name of the stored procedure. For example, if `proclib!func` was specified, then `proclib` would be loaded into memory and the function `func` from that library would be executed. This allows multiple functions to be placed in the same stored procedure library.

The stored procedure library is located in the directories or specified in the LIBPATH variable, as described in *procedure-name*.

absolute-path!function-name
The *absolute-path* specifies the complete path to the stored procedure library.

In a UNIX-based system, for example, if `/u/terry/proclib!func` was specified, then the stored procedure library `proclib` would be obtained from the directory `/u/terry` and the function `func` from that library would be executed.

In OS/2, if `d:\terry\proclib!func` was specified, then it would cause the database manager to load the func.dll file from the `d:\terry\proclib` directory.

In all these cases, the total length of the procedure name including its implicit or explicit full path must not be longer than 254 bytes.

DB2 Universal Database for OS/390 (V4.1 or later) server:
An implicit or explicit three part name. The parts are as follows.

high order: The location name of the server where the procedure is stored.

middle: SYSPROC

middle: Some value in the PROCEDURE column of the SYSIBM.SYSPROCEDURES catalog table.

DB2 for OS/400 (V3.1 or later) server:
The external program name is assumed to be the same as the *procedure-name*.

For portability, *procedure-name* should be specified as a single token no larger than 8 bytes.

(host-variable,...)
Each specification of *host-variable* is a parameter of the CALL. The nth parameter of the CALL corresponds to the nth parameter of the server's stored procedure.

Each *host-variable* is assumed to be used for exchanging data in both directions between client and server. In order to avoid sending unnecessary data between client and server, the client application should provide an indicator variable with each parameter and set the indicator to -1 if the parameter is not used to transmit data to the stored procedure. The stored procedure should set the indicator variable to -128 for any parameter that is not used to return data to the client application.

If the server is DB2 Universal Database the parameters must have matching data types in both the client and server program. [62]

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of host variables. The nth SQLVAR element corresponds to the nth parameter of the server's stored procedure.

Before the CALL statement is processed, the application must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables. The following fields of each Base SQLVAR element passed must be initialized:
  - SQLTYPE
  - SQLLEN
  - SQLDATA

62. DB2 Universal Database for OS/390 and DB2 Universal Database for AS/400 servers support conversions between compatible data types when invoking their stored procedures. For example, if the client program uses the INTEGER data type and the stored procedure expects FLOAT, the server will convert the INTEGER value to FLOAT before invoking the procedure.

– SQLIND

The following fields of each Secondary SQLVAR element passed must be initialized:

– LEN.SQLLONGLEN
– SQLDATALEN
– SQLDATATYPE_NAME

Each SQLDA is assumed to be used for exchanging data in both directions between client and server. In order to avoid sending unnecessary data between client and server, the client application should set the SQLIND field to -1 if the parameter is not used to transmit data to the stored procedure. The stored procedure should set the SQLIND field -128 for any parameter that is not used to return data to the client application.

Notes

- ***Use of Large Object (LOB) data types***:

  If the client and server application needs to specify LOB data from an SQLDA, allocate double the number of SQLVAR entries.

  LOB data types are supported by stored procedures starting with DB2 Version 2. The LOB data types are not supported by all down level clients or servers.

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

  If an SQL procedure successfully issues a RETURN statement with a status value, this value is returned in the first SQLERRD field of the SQLCA. If the CALL statement is issued in an SQL procedure, use the GET DIAGNOSTICS statement to retrieve the RETURN_STATUS value. The value is –1 if the SQLSTATE indicates an error.

- ***Returning Result Sets from Stored Procedures***:

  If the client application program is written using CLI, result sets can be returned directly to the client application. The stored procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on the result set, and leaving the cursor open when exiting the procedure.

  At the end of a procedure that is invoked via CLI:

  – For every cursor that has been left open, a result set is returned to the application.
  – If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened.
  – Only unread rows are passed back. For example, if the result set of a cursor has 500 rows, and 150 of those rows have been read by the stored procedure at the time the stored procedure is terminated, then rows 151 through 500 will be returned to the stored procedure.

For additional information refer to the *Application Development Guide* and the *CLI Guide and Reference*.

- ***Inter-operability between the CALL statement and the DARI API***:

  In general, the CALL statement will not work with existing DARI procedures. See the *Application Development Guide* for details.

- ***Handling of special registers***:

  The settings of the special registers of the caller are inherited by the stored procedure on invocation and restored upon return to the caller. Special registers may be changed within a stored procedure, but these changes do not effect the caller. This is not true for legacy stored procedures (those defined with parameter style DB2DARI or found in the default library), where the changes made to special registers in a procedure become the settings for the caller.

Examples

Example 1:

In C, invoke a procedure called TEAMWINS in the ACHIEVE library passing it a parameter stored in the host variable HV_ARGUMENT.

```
strcpy(HV_PROCNAME, "ACHIEVE!TEAMWINS");
CALL :HV_PROCNAME (:HV_ARGUMENT);
```

Example 2:

In C, invoke a procedure called :SALARY_PROC using the SQLDA named INOUT_SQLDA.

```
struct sqlda *INOUT_SQLDA;

/* Setup code for SQLDA variables goes here */

CALL :SALARY_PROC
USING DESCRIPTOR :*INOUT_SQLDA;
```

Example 3:

A Java stored procedure is defined in the database using the following statement:

```
CREATE PROCEDURE PARTS_ON_HAND (IN  PARTNUM  INTEGER,
                                OUT COST     DECIMAL(7,2),
                                OUT QUANTITY INTEGER)
                EXTERNAL NAME 'parts!onhand'
                LANGUAGE JAVA PARAMETER STYLE DB2GENERAL;
```

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

```
...
CallableStatement stpCall ;

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

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

stpCall.setInt( 1, variable1 ) ;
stpCall.setBigDecimal( 2, variable2 ) ;
stpCall.setInt( 3, variable3 ) ;

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

stpCall.execute() ;

variable2 = stpCall.getBigDecimal(2) ;
variable3 = stpCall.getInt(3) ;
...
```

This application code fragment will invoke the Java method *onhand* in class *parts* since the procedure-name specified on the CALL statement is found in the database and has the external name 'parts!onhand'.

CLOSE

The CLOSE statement closes a cursor. If a result table was created when the cursor was opened, that table is destroyed.

Invocation

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

Authorization

None required. See "DECLARE CURSOR" on page 841 for the authorization required to use a cursor.

Syntax

Description

cursor-name
Identifies the cursor to be closed. The *cursor-name* must identify a declared cursor as explained in the DECLARE CURSOR statement. When the CLOSE statement is executed, the cursor must be in the open state.

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

Notes

- At the end of a unit of work, all cursors that belong to an application process and that were declared without the WITH HOLD option are implicitly closed.
- CLOSE does not cause a commit or rollback operation.
- The WITH RELEASE clause has no effect for cursors that are operating under isolation levels CS or UR. When specified for cursors that are operating under isolation levels RS or RR, WITH RELEASE terminates some of the guarantees of those isolation levels. Specifically, if the cursor is opened again, an RS cursor may experience the 'nonrepeatable read' phenomenon and an RR cursor may experience either the 'nonrepeatable read' or 'phantom' phenomenon. Refer to "Appendix I. Comparison of Isolation Levels" on page 1285 for more details.

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

- Special rules apply to cursors within a stored procedure that have not been closed before returning to the calling program. See "Notes" on page 527 for more information.

Example

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

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO
   FROM TDEPT
   WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {                                                .
   EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
      .
      .
}

EXEC SQL CLOSE C1;
```

COMMENT ON

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

Invocation

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

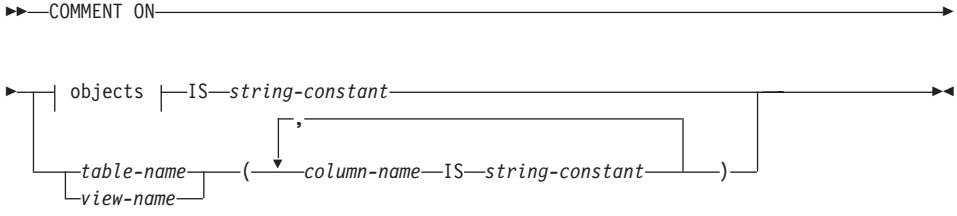
Authorization

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

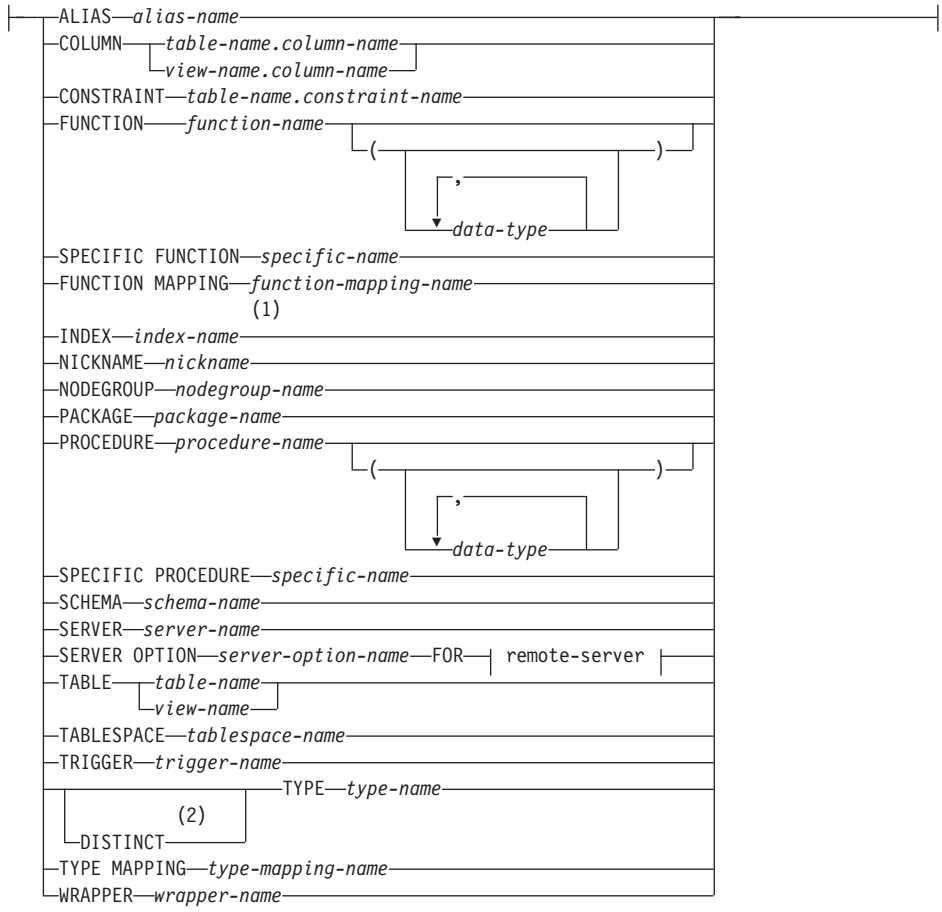
- SYSADM or DBADM
- definer of the object (underlying table for column or constraint) as recorded in the DEFINER column of the catalog view for the object (OWNER column for a schema)
- ALTERIN privilege on the schema (applicable only to objects allowing more than one-part names)
- CONTROL privilege on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

Note that for table space or nodegroup the authorization ID must have SYSADM or SYSCTRL authority.

Syntax

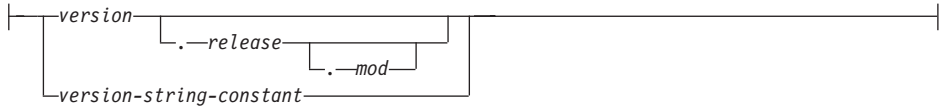


objects:



remote-server:

server-version:



Notes:

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

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

Description

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

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

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

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

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

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

FUNCTION *function-name*
Identifies the particular function, and is valid only if there is exactly one function with the *function-name*. The function thus identified may have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the function in the named or implied schema, an error (SQLSTATE 42854) is raised.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

PACKAGE *package-name*
Indicates a comment will be added or replaced for a package. The *package-name* must identify a distinct package that is described in the

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

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

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

PROCEDURE *procedure-name*
Identifies the particular procedure, and is valid only if there is exactly one procedure with the *procedure-name* in the schema. The procedure thus identified may have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the procedure in the named or implied schema, an error (SQLSTATE 42854) is raised.

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

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

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

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

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead an empty set of parentheses

may be coded to indicate that these attributes are to be ignored when looking for a data type match.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TABLE *table-name* or *view-name*
Indicates a comment will be added or replaced for a table or view. The *table-name* or *view-name* must identify a table or view (not an alias or nickname) that is described in the catalog (SQLSTATE 42704) and must

not identify a declared temporary table (SQLSTATE 42995). The comment replaces the value for the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the table or view.

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

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

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

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

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

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

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

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

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

Examples

Example 1: Add a comment for the EMPLOYEE table.

```
COMMENT ON TABLE EMPLOYEE
 IS 'Reflects first quarter reorganization'
```

Example 2: Add a comment for the EMP_VIEW1 view.

```
COMMENT ON TABLE EMP_VIEW1
 IS 'View of the EMPLOYEE table without salary information'
```

Example 3: Add a comment for the EDLEVEL column of the EMPLOYEE table.

```
COMMENT ON COLUMN EMPLOYEE.EDLEVEL
 IS 'highest grade level passed in school'
```

Example 4: Add comments for two different columns of the EMPLOYEE table.

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 11: Comment on an index specification named INDEXSPEC.

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

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

```
COMMENT ON WRAPPER NET8
   IS 'The wrapper for data sources associated with
       Oracle's Net8 client software.'
```

COMMIT

The COMMIT statement terminates a unit of work and commits the database changes that were made by that unit of work.

Invocation

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

Authorization

None required.

Syntax

Description

The unit of work in which the COMMIT statement is executed is terminated and a new unit of work is initiated. All changes made by the following statements executed during the unit of work are committed: ALTER, COMMENT ON, CREATE, DELETE, DROP, GRANT, INSERT, LOCK TABLE, REVOKE, SET INTEGRITY, SET transition-variable, and UPDATE.

The following statements, however, are not under transaction control and changes made by them are independent of issuing the COMMIT statement:

- SET CONNECTION,
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE,
- SET CURRENT EXPLAIN MODE,
- SET CURRENT EXPLAIN SNAPSHOT,
- SET CURRENT PACKAGESET,
- SET CURRENT QUERY OPTIMIZATION,
- SET CURRENT REFRESH AGE,
- SET EVENT MONITOR STATE,
- SET PASSTHRU,
- SET PATH,
- SET SCHEMA,
- SET SERVER OPTION.

All locks acquired by the unit of work subsequent to its initiation are released, except necessary locks for open cursors that are declared WITH HOLD.All open cursors not defined WITH HOLD are closed. Open cursors defined

WITH HOLD remain open, and the cursor is positioned before the next logical row of the result table.[63] All LOB locators are freed. Note that this is true even when the locators are associated with LOB values retrieved via a cursor that has the WITH HOLD property.

All savepoints set within the transaction are released.

Notes

- It is strongly recommended that each application process explicitly ends its unit of work before terminating. If the application program ends normally without a COMMIT or ROLLBACK statement then the database manager attempts a commit or rollback depending on the application environment. Refer to *Application Development Guide* for implicitly ending a transaction in different application environments.
- See "EXECUTE" on page 895 for information on the impact of COMMIT on cached dynamic SQL statements.
- See "DECLARE GLOBAL TEMPORARY TABLE" on page 846 for information on potential impacts of COMMIT on declared temporary tables.

Example

Commit alterations to the database made since the last commit point.

```
COMMIT WORK
```

63. A FETCH must be performed before a Positioned UPDATE or DELETE statement is issued.

Compound SQL (Embedded)

Combines one or more other SQL statements (*sub-statements*) into an executable block. Please see "Chapter 7. SQL Procedures" on page 1059 for Compound SQL statements within SQL procedures.

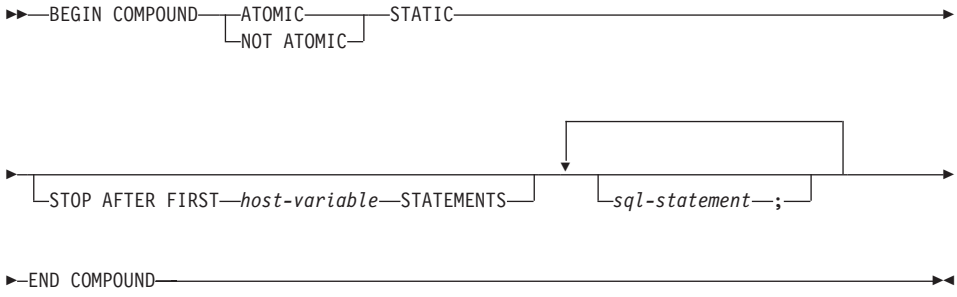
Invocation

This statement can only be embedded in an application program. The entire Compound SQL statement construct is an executable statement that cannot be dynamically prepared. The statement is not supported in REXX.

Authorization

None for the Compound SQL statement itself. The authorization ID of the Compound SQL statement must have the appropriate authorization on all the individual statements that are contained within the Compound SQL statement.

Syntax



Description

ATOMIC
Specifies that, if any of the sub-statements within the Compound SQL statement fail, then all changes made to the database by any of the sub-statements, including changes made by successful sub-statements, are undone.

NOT ATOMIC
Specifies that, regardless of the failure of any sub-statements, the Compound SQL statement will not undo any changes made to the database by the other sub-statements.

STATIC
Specifies that input variables for all sub-statements retain their original value. For example, if

```
SELECT ... INTO :abc ...
```

is followed by:

```
UPDATE T1 SET C1 = 5 WHERE C2 = :abc
```

the UPDATE statement will use the value that :abc had at the start of the execution of the Compound SQL statement, not the value that follows the SELECT INTO.

If the same variable is set by more than one sub-statement, the value of that variable following the Compound SQL statement is the value set by the last sub-statement.

Note: Non-static behavior is not supported. This means that the sub-statements should be viewed as executing non-sequentially and sub-statements should not have interdependencies.

STOP AFTER FIRST
Specifies that only a certain number of sub-statements will be executed.

host-variable
A small integer that specifies the number of sub-statements to be executed.

STATEMENTS
Completes the STOP AFTER FIRST *host-variable* clause.

sql-statement
All executable statements except the following can be contained within an embedded static compound SQL statement:

CALL
CLOSE
CONNECT
Compound SQL
DESCRIBE
DISCONNECT
EXECUTE IMMEDIATE
FETCH
OPEN
PREPARE
RELEASE (Connection)
RELEASE SAVEPOINT
ROLLBACK
SAVEPOINT
SET CONNECTION

If a COMMIT statement is included, it must be the last sub-statement. If COMMIT is in this position, it will be issued even if the STOP AFTER FIRST *host-variable* STATEMENTS clause indicates that not all of the sub-statements are to executed. For example, suppose COMMIT is the last sub-statement in a compound SQL block of 100 sub-statements. If the STOP AFTER FIRST STATEMENTS clause indicates that only 50 sub-statements are to be executed, then COMMIT will be the 51st sub-statement.

An error will be returned if COMMIT is included when using CONNECT TYPE 2 or running in an XA distributed transaction processing environment (SQLSTATE 25000).

Rules

- DB2 Connect does not support SELECT statements selecting LOB columns in a compound SQL block.
- No host language code is allowed within a Compound SQL statement; that is, no host language code is allowed between the sub-statements that make up the Compound SQL statement.
- Only NOT ATOMIC Compound SQL statements will be accepted by DB2 Connect.
- Compound SQL statements cannot be nested.
- An Atomic Compound SQL statement cannot be issued inside a savepoint (SQLSTATE 3B002).

Notes

One SQLCA is returned for the entire Compound SQL statement. Most of the information in that SQLCA reflects the values set by the application server when it processed the last sub-statement. For instance:

- The SQLCODE and SQLSTATE are normally those for the last sub-statement (the exception is described in the next point).
- If a 'no data found' warning (SQLSTATE '02000') is returned, then that warning is given precedence over any other warning in order that a WHENEVER NOT FOUND exception can be acted upon.[64]
- The SQLWARN indicators are an accumulation of the indicators set for all sub-statements.

If one or more errors occurred during NOT ATOMIC Compound SQL execution and none of these are of a serious nature, the SQLERRMC will contain information on up to a maximum of seven of these errors. The first token of the SQLERRMC will indicate the total number of errors that occurred. The remaining tokens will each contain the ordinal position and the SQLSTATE of the failing sub-statement within the Compound SQL statement. The format is a character string of the form:

```
nnnXsssccccc
```

64. This means that the SQLCODE, SQLERRML, SQLERRMC, and SQLERRP fields in the SQLCA that is eventually returned to the application are those from the sub-statement that triggered the 'no data found'. If there is more than one 'no data found' warning within the Compound SQL statement, the fields for the last sub-statement will be the fields returned.

with the substring starting with X repeating up to six more times and the string elements defined as follows.

nnn The total number of statements that produced errors. [65] This field is left-justified and padded with blanks.

X The token separator X'FF'.

sss The ordinal position of the statement that caused the error. [65] For example, if the first statement failed, this field would contain the number one left-justified ('1 ').

ccccc The SQLSTATE of the error.

The second SQLERRD field contains the number of statements that failed (returned negative SQLCODEs).

The third SQLERRD field in the SQLCA is an accumulation of the number of rows affected by all sub-statements.

The fourth SQLERRD field in the SQLCA is a count of the number of successful sub-statements. If, for example, the third sub-statement in a Compound SQL statement failed, the fourth SQLERRD field would be set to 2, indicating that 2 sub-statements were successfully processed before the error was encountered.

The fifth SQLERRD field in the SQLCA is an accumulation of the number of rows updated or deleted due to the enforcement of referential integrity constraints for all sub-statements that triggered such constraint activity.

Examples

Example 1: In a C program, issue a Compound SQL statement that updates both the ACCOUNTS and TELLERS tables. If there is an error in any of the statements, undo the effect of all statements (ATOMIC). If there are no errors, commit the current unit of work.

```
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
  UPDATE ACCOUNTS SET ABALANCE = ABALANCE + :delta
    WHERE AID = :aid;
  UPDATE TELLERS SET TBALANCE = TBALANCE + :delta
    WHERE TID = :tid;
  INSERT INTO TELLERS (TID, BID, TBALANCE) VALUES (:i, :branch_id, 0);
  COMMIT;
END COMPOUND;
```

65. If the number would exceed 999, counting restarts at zero.

Example 2: In a C program, insert 10 rows of data into the database. Assume the host variable :nbr contains the value 10 and S1 is a prepared INSERT statement. Further, assume that all the inserts should be attempted regardless of errors (NOT ATOMIC).

```
EXEC SQL BEGIN COMPOUND NOT ATOMIC STATIC STOP AFTER FIRST :nbr STATEMENTS
    EXECUTE S1 USING DESCRIPTOR :*sqlda0;
    EXECUTE S1 USING DESCRIPTOR :*sqlda1;
    EXECUTE S1 USING DESCRIPTOR :*sqlda2;
    EXECUTE S1 USING DESCRIPTOR :*sqlda3;
    EXECUTE S1 USING DESCRIPTOR :*sqlda4;
    EXECUTE S1 USING DESCRIPTOR :*sqlda5;
    EXECUTE S1 USING DESCRIPTOR :*sqlda6;
    EXECUTE S1 USING DESCRIPTOR :*sqlda7;
    EXECUTE S1 USING DESCRIPTOR :*sqlda8;
    EXECUTE S1 USING DESCRIPTOR :*sqlda9;
END COMPOUND;
```